

5.1.5 Assess the Current Codebase

When you inherit an unfamiliar codebase, take a few moments to familiarize yourself with the current state of the project. You don't have to understand every line of code, though. Sometimes developers inherit projects thousands of lines long! What's important is gaining a general sense of how the app works and taking a deeper look at the code blocks that apply to the feature or bug you're working on.

ON THE JOB

Your first task as a paid developer will most likely involve an existing project.

Companies rarely ask their new developers to build something from scratch. You can expect that the company will give you time to read, experiment with, and test the legacy code before they expect you to start changing it.

First, open the `assets/js/script.js` file in VS Code and skim over the code. Don't freak out when you see a lot of unfamiliar syntax. For example:

```
// modal was triggered
$("#task-form-modal").on("show.bs.modal", function() {
```

```
// clear values
$("#modalTaskDescription, #modalDueDate").val("");
});

// modal is fully visible
$("#task-form-modal").on("shown.bs.modal", function() {
  // highlight textarea
  $("#modalTaskDescription").trigger("focus");
});
```

Why are there `<script>` characters everywhere? Open the `index.html` file in VS Code and scroll to the bottom. You'll notice several `<script>` elements:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.15.0/umd/popper.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.5.0/js/bootstrap.min.js"></script>
<script src="./assets/js/script.js"></script>
```

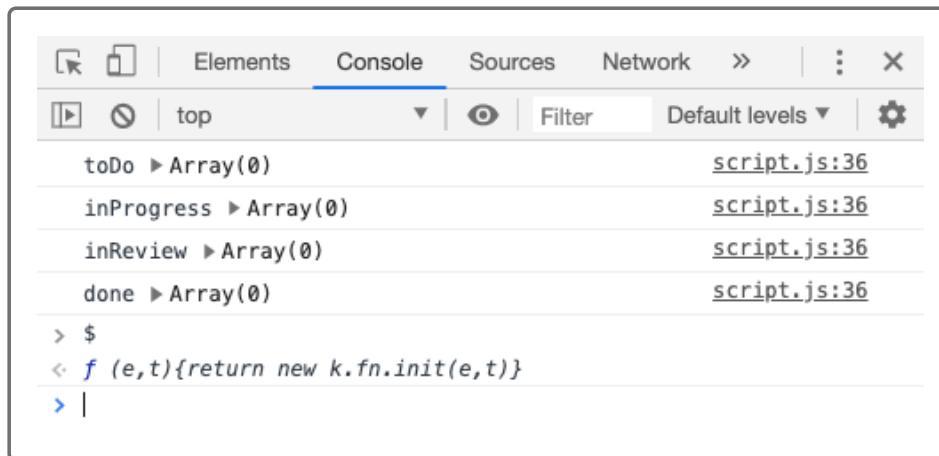
Only one of those files, `script.js`, is ours. The rest come from an external source. This is an example of a **Content Delivery Network**, or **CDN**. A CDN hosts files as a public service to make loading these resources faster and easier.

If you visit one of those URLs in the browser, you'll see that they are indeed JavaScript files, just a little messier than you might be used to. You will see code that looks like the following image:



By including the CDN URL in the HTML, we can load all the functionality (methods, objects, etc.) that are predefined in this file. In the case of `jquery.min.js`, this file provides a very powerful function represented by a `$` character.

If you haven't already, open `index.html` in Chrome to bring up the Taskmaster app. Open the Chrome DevTools and try accessing this `$` function in the console, as demonstrated in the image below:



IMPORTANT

Feel free to remove the `console.log(list, arr);` in the JavaScript code you downloaded, this is more to make sure it worked when we inherited it.

On that note, feel free to remove `console.log()` statements we create throughout projects, as old ones can clutter the DevTools Console as we debug new code in our apps.

We can't figure out the `$` function's usefulness just by looking at it. We're better off consulting the official documentation for the jQuery library.

In the browser, navigate to [jQuery.com](https://jquery.com/) `(https://jquery.com/)` and look over some of their examples. The jQuery homepage shows an example of

route you to the [documentation for the append\(\) method](https://api.jquery.com/append/) (<https://api.jquery.com/append/>). This page goes into greater detail on how to use it, as shown in the image below:

Consider the following HTML:

```
1 | <h2>Greetings</h2>
2 | <div class="container">
3 |   <div class="inner">Hello</div>
4 |   <div class="inner">Goodbye</div>
5 | </div>
```

You can create content and insert it into several elements at once:

```
1 | $( ".inner" ).append( "<p>Test</p>" );
```

Each inner `<div>` element gets this new content:

```
1 | <h2>Greetings</h2>
2 | <div class="container">
3 |   <div class="inner">
4 |     Hello
5 |     <p>Test</p>
6 |   </div>
7 |   <div class="inner">
8 |     Goodbye
9 |     <p>Test</p>
10 |   </div>
11 | </div>
```

At a glance, jQuery's `append()` is essentially a more robust shorthand for `appendChild()`. With plain JavaScript, you'd have to iterate over multiple DOM elements and call `appendChild()` on each one. jQuery's `append()` does that behind the scenes, allowing your selector to be more generic (e.g., `".inner"`).

We'll have a chance to practice `append()` and other jQuery methods soon enough. Let's turn our attention to the styling of Taskmaster. The webpage looks good already, but if you open the `style.css` file, you'll see it is pretty bare:

```
ul.list-group {
  min-height: 130px;
}
```

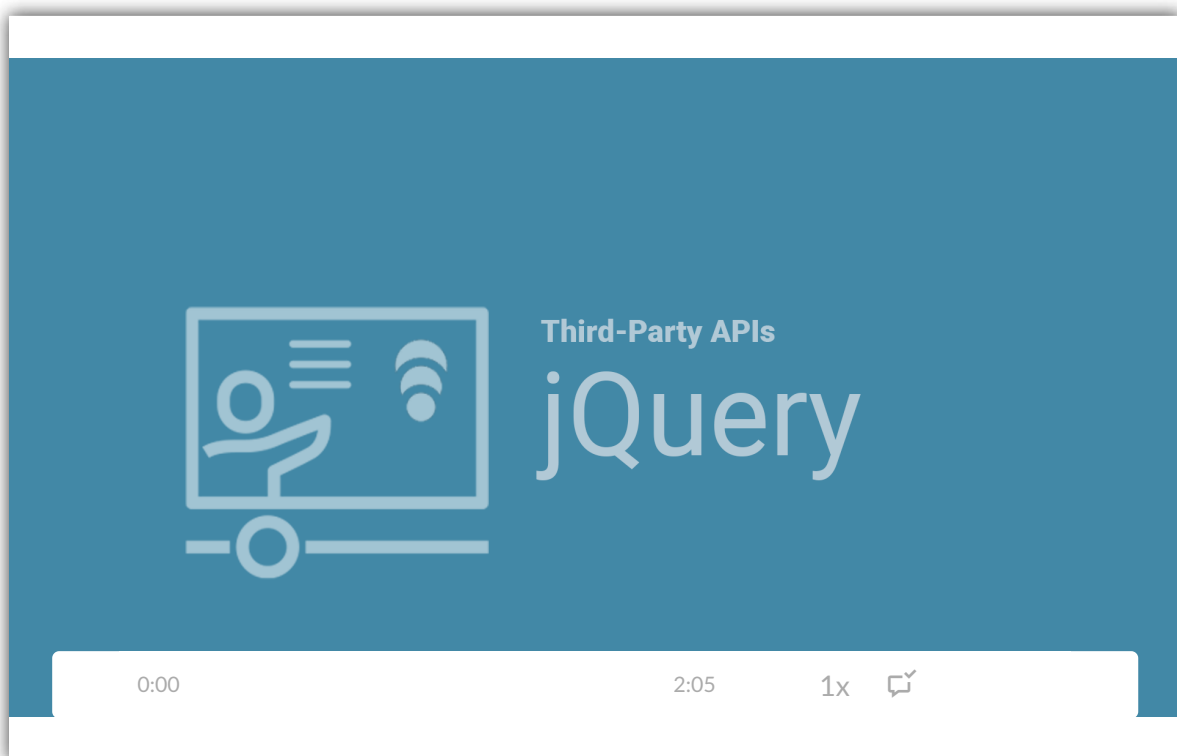
changing a button's content with the `$` shorthand, but regular JavaScript methods can accomplish the same thing, as shown here:

```
// jquery version
$( "button.continue" ).html( "Next Step..." );

// plain javascript
document.querySelector("button.continue").innerHTML = "Next Step...";
```

Ultimately, jQuery provides shorter methods to find and manipulate DOM elements, add event listeners, and perform other common JavaScript-related tasks. Plain JavaScript can do everything jQuery does, but many developers find the jQuery syntax easier to remember and type.

For more information on jQuery, watch the following video:



From the jQuery homepage, navigate to the [API Documentation](https://api.jquery.com/) (<https://api.jquery.com/>) tab and scroll through the many shorthand methods that jQuery provides. Clicking on the `append()` method name will

Refer back to the `index.html` file and look at the `<head>` element. Two style sheets are actually being loaded:

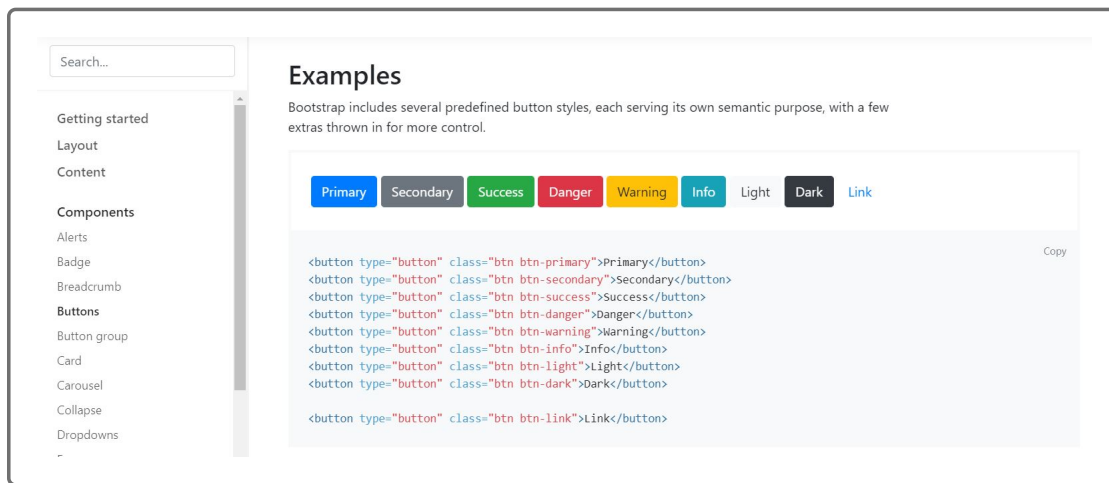
```
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.3.1/css/bootstrap.min.css">
<link rel="stylesheet" href="./assets/css/style.css" />
```

One style sheet is ours (`style.css`), but the other comes from a CDN again. Taskmaster is using another style sheet, from a library called **Bootstrap**. Like jQuery, the best way to understand Bootstrap is to read the documentation.

NERD NOTE

Did you notice the directory `twitter-bootstrap` in the CDN URL? The company Twitter developed Bootstrap, then gave it to the public as open source software.

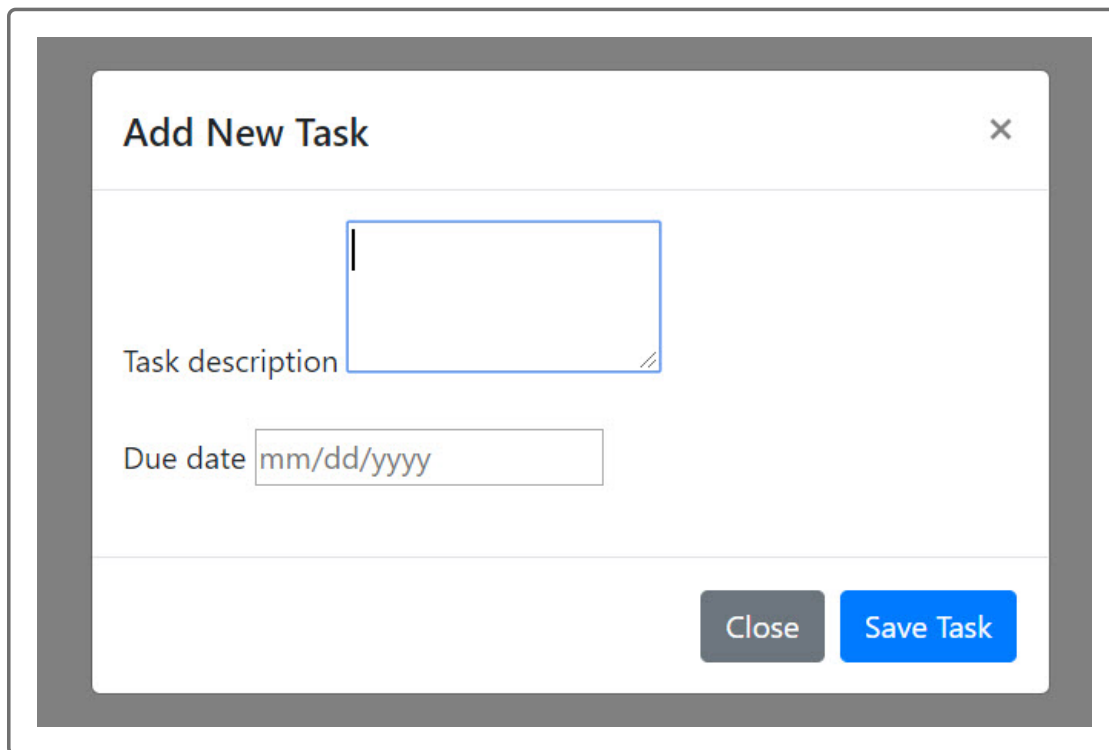
In the browser, visit the [Bootstrap documentation website](https://getbootstrap.com/docs/4.3/getting-started/introduction/) (<https://getbootstrap.com/docs/4.3/getting-started/introduction/>) and click through some of the navigation links on the left. You'll see many examples of HTML code snippets that become colorful elements like buttons, as shown in the following screenshot:



In the previous example, Bootstrap demonstrates adding a `"btn btn-primary"` class attribute to a `<button>` element to make it blue. Bootstrap predefines the classes `btn` and `btn-primary`, but by including the Bootstrap style sheet in a given project, you can use the same class names to quickly style similar-looking UI elements.

Bootstrap also offers many class names that speed up building responsive layouts. We'll dive deeper into Bootstrap's layout classes in the next lesson. For now, we should focus on the workflow that precedes editing a task (the missing feature).

Open the `index.html` file in Chrome and click the Add Task button. This will display a modal where the user can enter a task description and due date. A **modal** (sometimes called a **pop-up**) is an overlay that requires user interaction before returning to the main application, as seen in this screenshot from Taskmaster:

A modal window titled "Add New Task" with a close button (X) in the top right corner. Inside the modal, there is a text input field for "Task description" and a date input field for "Due date" with a placeholder "mm/dd/yyyy". At the bottom right of the modal, there are two buttons: "Close" and "Save Task".

Add New Task ×

Task description

Due date

Close Save Task

After the user fills out the Add New Task form in the modal, they can click the Save Changes button to add the task to the list. The following code block in `script.js` captures that button click:

```
$("#task-form-modal .btn-primary").click(function() {  
  });
```

PAUSE

How would you attach a click event listener to an element with plain JavaScript?

```
document.querySelector("#task-form-modal .btn-primary").addl  
  });
```

[Hide Answer](#)

Inside the click function, a call is made to `createTask(taskText, taskDate, "todo")`, passing in the task's description, due date, and type (hardcoded as `"todo"`). In the `createTask()` function, these three data points create a `` element (with child `` and `<p>` elements) that's appended to a `` element:

```
var createTask = function(taskText, taskDate, taskList) {  
  // create elements that make up a task item  
  var taskLi = $("- ").addClass("list-group-item");  
  
  var taskSpan = $("")  
    .addClass("badge badge-primary badge-pill")  
    .text(taskDate);  
  
  var taskP = $("

")  
    .addClass("m-1")  
    .text(taskText);  
  
  // append span and p element to parent li  
  taskLi.append(taskSpan, taskP);  
  
  // append to ul list on the page  
  $("#list-" + taskList).append(taskLi);  
};

```

Note the use of jQuery methods like `addClass()` and `text()` and Bootstrap classes like `list-group-item` and `badge`. The `addClass()` method updates an element's `class` attribute with new values while the `text()` method changes the element's text content (similar to `element.textContent = "";` in plain JavaScript).

It's okay if these methods and classes are confusing right now. They're new, so they should feel that way! You can always look up their meaning in the documentation or practice them in a different context.

Luckily, this function already works, so we don't need to spend a lot of time dissecting it. We can move on to the next function call: `saveTasks()`. This is called after `createTask()` in the click function, shown in the following code block:

```
tasks.todo.push({
  text: taskText,
  date: taskDate
});

saveTasks();
```

The `saveTasks()` function simply saves the `tasks` object in `localStorage`, as we can see here:

```
var saveTasks = function() {
  localStorage.setItem("tasks", JSON.stringify(tasks));
};
```

Note that tasks are saved in an array that's a property of an object. The previous Taskmaster developer had the foresight to structure the data this way, knowing multiple lists would be added at some point. You can see this more easily in the following `loadTasks()` function:

```
var loadTasks = function() {
  tasks = JSON.parse(localStorage.getItem("tasks"));

  // if nothing in localStorage, create a new object to track all task
  if (!tasks) {
    tasks = {
      todo: [],
      inProgress: [],
      inReview: [],
      done: []
    };
  };
};
```

```
}  
};
```

Eventually, we'll use all of these arrays, but for now let's focus on getting and pushing tasks to and from `tasks.todo`.

By now, we hope you feel a real sense of accomplishment—that's a lot of logic already in place! Don't worry if you don't understand every detail of how this works, though. As long as you have a general idea of the workflow and know where to find help (by reading the documentation), you're right on target. Let's start on that editing feature now.

© 2020 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.