

4.2.8 Address Usability Concerns

The end of the last lesson may have gotten you on the edge of your seat wondering how the code could possibly be broken, so here goes.

What happens if we submit the form without filling anything out? What happens if we put in a task name but forget to pick a task type? Try doing it. You should get this result:



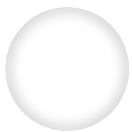
As the above image shows, the task item was created just fine, but it's missing content! We can't delete a task (yet), but removing empty tasks would be cumbersome even with that ability. We should validate the form fields before completing the submission.

Validation comes in a lot of different flavors, but it typically means checking that the required form fields have content and the content fits our needs. Think about creating a password: an application usually won't permit the creation of a short password without a number or uppercase letter.

Luckily we don't have to test for that here. We just need to check whether the form fields have content or not. If they do, let the function continue and create the task item. If either field doesn't, stop the function and let the user know that something is missing.

Let's do this right now and update `taskFormHandler()`, to have this code right before we create the `taskDataObj` variable:

```
// check if input values are empty strings
if (!taskNameInput || !taskTypeInput) {
  alert("You need to fill out the task form!");
  return false;
}
```



REWIND

When used in a condition, empty strings and the number 0 are evaluated as **falsy values**. When we use the syntax `!taskNameInput`, we're checking to see if the `taskNameInput` variable is empty by asking if it's a falsy value.

That's what the "not" operator `!` is doing here. Because the default is to check for a true value, the `!` is used to check for the opposite (false) value.

If we save the `script.js` file and try to submit a task with an empty field now, we'll receive an alert that tells us we need to fill out the form, and then the function will stop when it reads `return false`. But if we include data in the form fields, the `taskFormHandler()` will work as intended and send the data to `createTaskEl()` to be printed to the page.

This is a basic version of form input validation; we simply check to see if any value is read from the form inputs. By using the condition below, we're seeing if either `taskNameInput` or `taskTitleInput` is empty, or if both are empty.

```
(!taskNameInput || !taskTypeInput)
```

Putting an exclamation point (!) in front of the variable name will make the condition return true if the value evaluates to false. So, this code literally says, "if either one or both of the variables are *not* true, then proceed," which is the same as "if either one or both of the variables are false, then proceed."

That seems confusing at first, but think of it this way: we're checking to see if a `false` value is in fact `false`, which would result in the condition being `true`. Let's explore some examples of this before moving on:

```
if (true) {  
  // this will run because true is true  
  console.log("Is true true? Yes.");  
}  
  
if (false) {  
  // this will not run because false is not true  
  console.log("Is false true? No.");  
}  
  
if (3 === 10 || "a" === "a") {  
  // this will run because at least one of the conditions is true  
  console.log("Does 3 equal 10? No.");  
  console.log("Does the letter 'a' equal the letter 'a'? Yes.");  
}
```

```
}  
  
if (3 === 10 && "a" === "a") {  
  // this will not run because both conditions have to be true to run  
  console.log("Does 3 equal 10? No.");  
  console.log("Does the letter 'a' equal the letter 'a'? Yes.");  
}
```

We've now fixed a fairly large issue this application could have, and we did it with only a few lines of code! We've made the application more user-friendly by making sure those two variables aren't empty strings and by giving feedback if users make a mistake.

Reset the Form

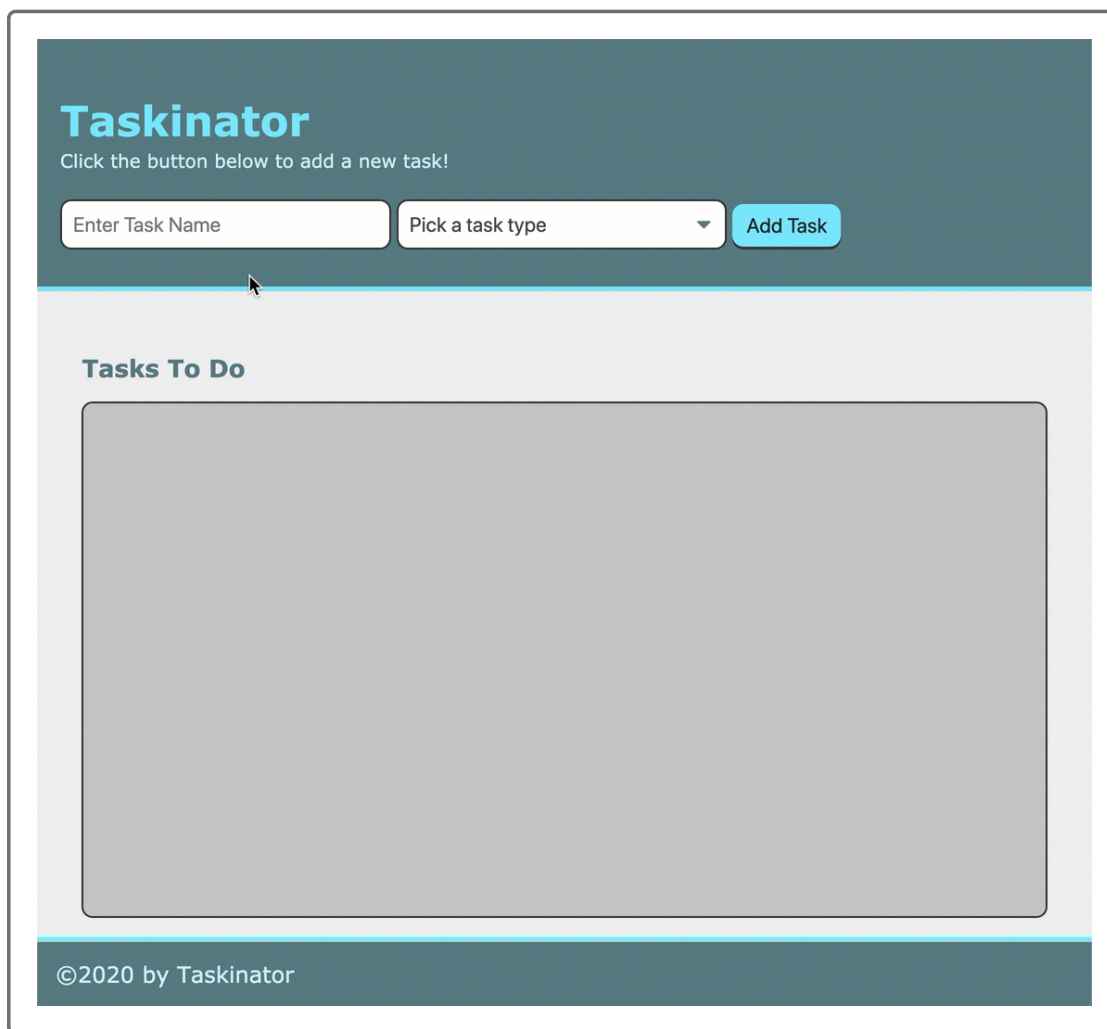
We have one more thing to fix, however, and it's not so much an issue as a usability enhancement.

Let's say we're adding multiple tasks at once. Every time we fill out the form and submit a new task, we have to erase the previous task's content and start over again. While this isn't a breaking bug or issue, it adds a little frustration for the user. Luckily, we can fix this frustration with one line of code.

Let's add this one last line of code to `taskFormHandler()`, underneath the validation `if` statement we just added:

```
formEl.reset();
```

Save `script.js` and try to submit a task. Note that the form resets itself to its default values when you successfully submit a task. Before, it retained the task name and task type after you submitted the form.



The screenshot shows a web application titled "Taskinator". At the top, there is a dark teal header with the title "Taskinator" in white. Below the title, a message says "Click the button below to add a new task!". There are two input fields: "Enter Task Name" and "Pick a task type" (a dropdown menu). To the right of these fields is a blue "Add Task" button. Below the header is a light gray section titled "Tasks To Do". Inside this section is a large, empty gray rectangular box, likely a placeholder for a list of tasks. At the bottom of the page is a dark teal footer with the text "©2020 by Taskinator".

The browser-provided DOM element interface has the `reset()` method, which is designed specifically for the `<form>` element and won't work on any other element. We could use other methods, namely by targeting each form element and resetting their values manually, but that could be cumbersome if resetting a larger form.

DEEP DIVE ▲

DEEP DIVE

For more information, see the [MDN web docs on a form element's reset\(\) method](https://developer.mozilla.org/en-US/docs/Web/API/Form/reset)

[US/docs/Web/API/HTMLFormElement/reset](#)).

We're all set! We covered a lot of important ground in this lesson, and it shows. The Taskinator application not only accepts custom values from form elements but also validates the form as well. This is a big step in building this application, so let's close this feature branch's issue and merge it into the `develop` branch.

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