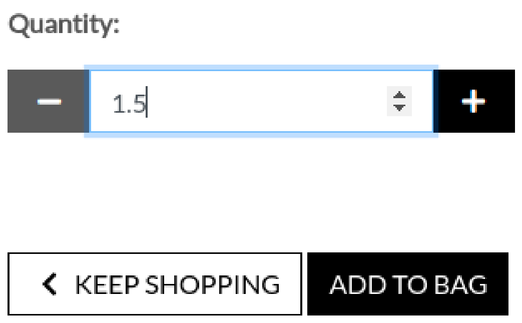
DEBUGGING - A DETECTIVE STORY AND A LEARNING EXPERIENCE

One morning, a student came into Tutoring with the following:

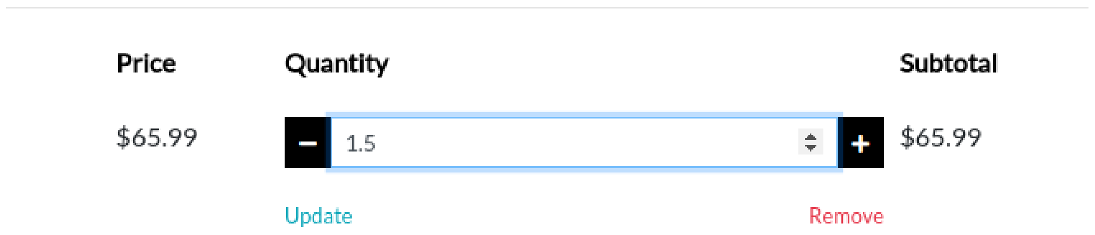
"In my e-commerce project, the quantity form in product_detail.html doesn't allow the input of floats (as is expected), but in bag.html it does, so when submitted there, it triggers a Django error. My code seems identical to the code used in Boutique Ado. I cannot seem to spot the error that is causing this."

As per the usual procedure, the Tutor - let's call him Peter - asked the student to share their Gitpod workspace. Once shared, Peter went in, opened the app, and tested the feature. And indeed, this is what he saw:

product_detail.html:

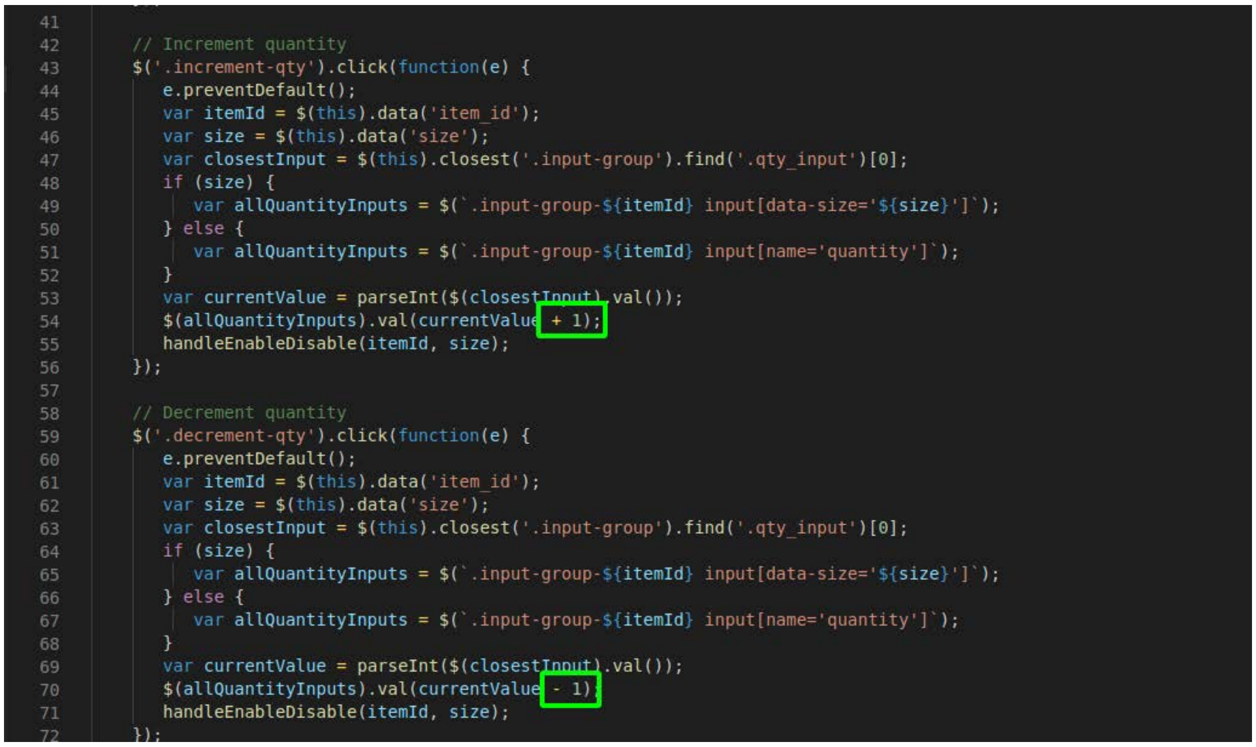


bag.html



So, in both places it was possible to *manually* enter a float.

The **+/-** buttons would, of course, only increment/decrement the value by a whole number, because that is how the quantity input script for the buttons was written:



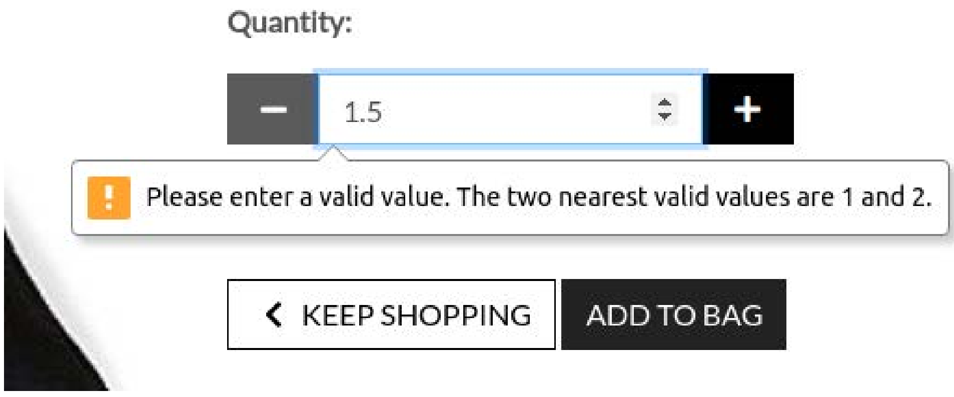
```
    // Increment quantity
    $('.increment-qty').click(function(e) {
        e.preventDefault();
        var itemId = $(this).data('item_id');
        var size = $(this).data('size');
        var closestInput = $(this).closest('.input-group').find('.qty_input')[0];
        if (size) {
            var allQuantityInputs = $(`.input-group-${itemId} input[data-size='${size}']`);
        } else {
            var allQuantityInputs = $(`.input-group-${itemId} input[name='quantity']`);
        }
        var currentValue = parseInt($(closestInput).val());
        $(allQuantityInputs).val(currentValue + 1);
        handleEnableDisable(itemId, size);
    });

    // Decrement quantity
    $('.decrement-qty').click(function(e) {
        e.preventDefault();
        var itemId = $(this).data('item_id');
        var size = $(this).data('size');
        var closestInput = $(this).closest('.input-group').find('.qty_input')[0];
        if (size) {
            var allQuantityInputs = $(`.input-group-${itemId} input[data-size='${size}']`);
        } else {
            var allQuantityInputs = $(`.input-group-${itemId} input[name='quantity']`);
        }
        var currentValue = parseInt($(closestInput).val());
        $(allQuantityInputs).val(currentValue - 1);
        handleEnableDisable(itemId, size);
    });
```

so there was nothing wrong with the script.

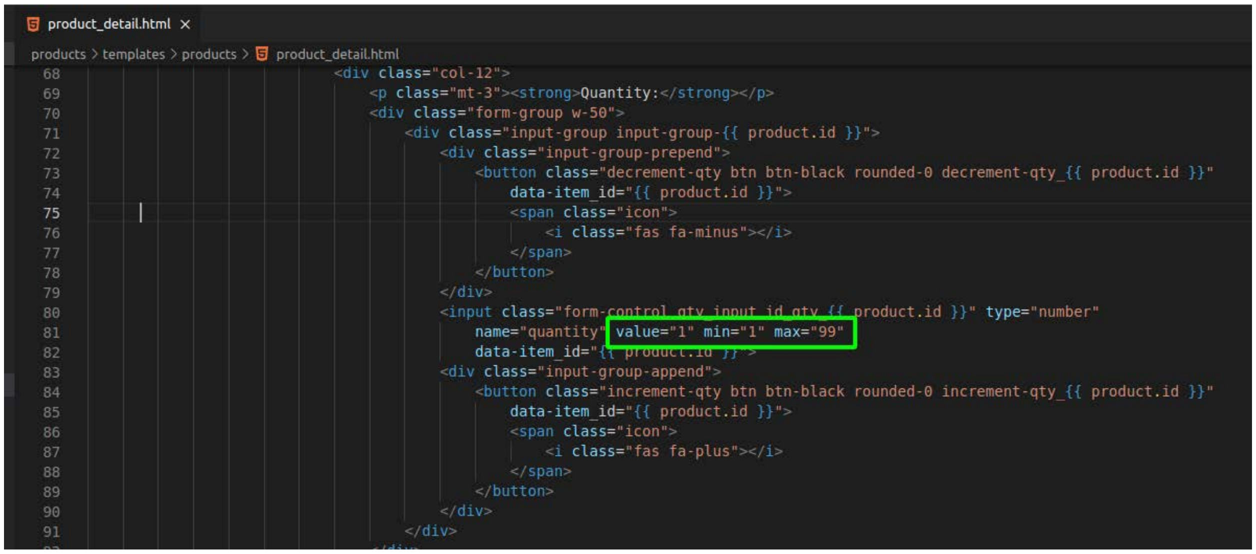
But the user could still input a float via the keyboard.

That, however, was not the problem in and of itself. When he tried to submit (**Add to Bag**) in **product_detail.html**, Peter got this:



Please enter a valid value. The two nearest valid values are 1 and 2.

which, of course, is precisely what he would have expected. The **`min=""`** and **`max=""`** values on the input field are set to 1 and 99 respectively. With no **`step=""`** explicitly specified by the code, the default step is set to 1. Thus the form only accepts increments of 1 starting from 1 - so integers only.



product_detail.html ×

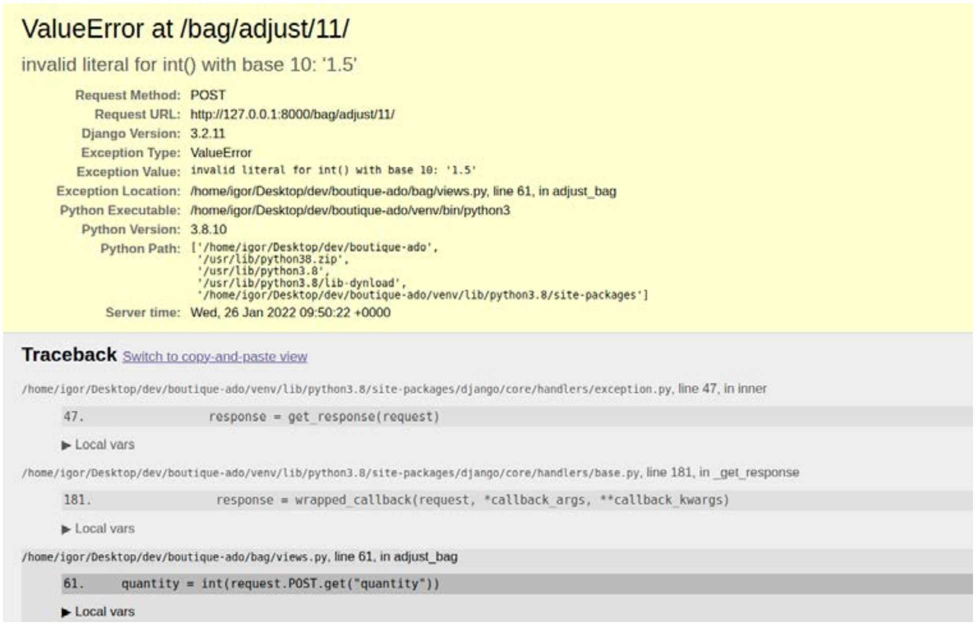
products > templates > products > product_detail.html

```html
<div class="col-12">
    <p class="mt-3"><strong>Quantity:</strong></p>
    <div class="form-group w-50">
        <div class="input-group input-group-{{ product.id }}">
            <div class="input-group-prepend">
                <button class="decrement-qty btn btn-black rounded-0 decrement-qty_{{ product.id }}"
                    data-item_id="{{ product.id }}">
                    <span class="icon">
                        <i class="fas fa-minus"></i>
                    </span>
                </button>
            </div>
            <input class="form-control qty_input id_qty_{{ product.id }}" type="number"
                name="quantity" value="1" min="1" max="99"
                data-item_id="{{ product.id }}">
            <div class="input-group-append">
                <button class="increment-qty btn btn-black rounded-0 increment-qty_{{ product.id }}"
                    data-item_id="{{ product.id }}">
                    <span class="icon">
                        <i class="fas fa-plus"></i>
                    </span>
                </button>
            </div>
        </div>
```

So any attempt to submit the form with a float is rejected by form validation.

Everything as expected.

However, in **bag.html**, when Peter input a float into the quantity form, and clicked the **Update** button/link, the form submitted, and he got this:



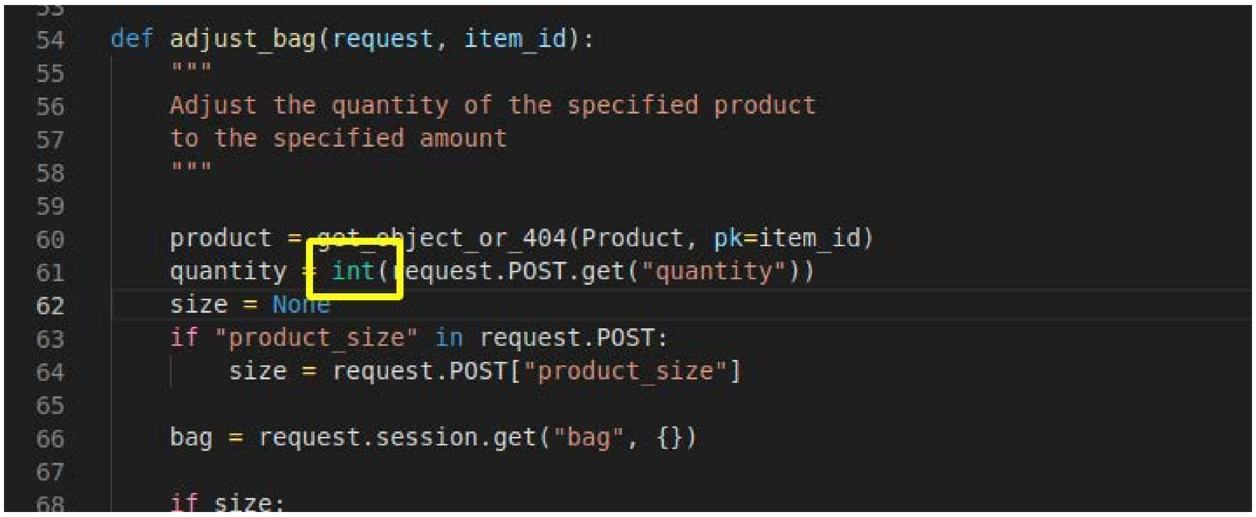
```
ValueError at /bag/adjust/11/
invalid literal for int() with base 10: '1.5'

      Request Method: POST
         Request URL: http://127.0.0.1:8000/bag/adjust/11/
      Django Version: 3.2.11
      Exception Type: ValueError
     Exception Value: invalid literal for int() with base 10: '1.5'
  Exception Location: /home/igor/Desktop/dev/boutique-ado/bag/views.py, line 61, in adjust_bag
   Python Executable: /home/igor/Desktop/dev/boutique-ado/venv/bin/python3
      Python Version: 3.8.10
         Python Path: ['/home/igor/Desktop/dev/boutique-ado',
                       '/usr/lib/python38.zip',
                       '/usr/lib/python3.8',
                       '/usr/lib/python3.8/lib-dynload',
                       '/home/igor/Desktop/dev/boutique-ado/venv/lib/python3.8/site-packages']
         Server time: Wed, 26 Jan 2022 09:50:22 +0000

Traceback Switch to copy-and-paste view

/home/igor/Desktop/dev/boutique-ado/venv/lib/python3.8/site-packages/django/core/handlers/exception.py, line 47, in inner
    47.                 response = get_response(request)
    ▶ Local vars
/home/igor/Desktop/dev/boutique-ado/venv/lib/python3.8/site-packages/django/core/handlers/base.py, line 181, in _get_response
    181.                 response = wrapped_callback(request, *callback_args, **callback_kwargs)
    ▶ Local vars
/home/igor/Desktop/dev/boutique-ado/bag/views.py, line 61, in adjust_bag
    61.     quantity = int(request.POST.get("quantity"))
    ▶ Local vars
```

So, the form in bag.html had submitted with a float input. The result was the Django error shown above, which was only completely logical, because the backend processing of the submitted form's quantity field (in the adjust_bag view) always expected an *integer-containing string*, not a float-containing one:



```
def adjust_bag(request, item_id):
    """
    Adjust the quantity of the specified product
    to the specified amount
    """

    product = get_object_or_404(Product, pk=item_id)
    quantity = int(request.POST.get("quantity"))
    size = None
    if "product_size" in request.POST:
        size = request.POST["product_size"]

    bag = request.session.get("bag", {})

    if size:
```

Peter was aware that, at this point, one could simply argue that nobody was going to seriously try to buy two-and-a-half T-shirts or half a bed linen. Still, the input was *possible*, it was accepted by the form, and it led to the app error, thus presenting a defensive weakness in the UI design - one that definitely needed to be addressed.

But why didn't the bag form simply reject the input like the product_detail form did?

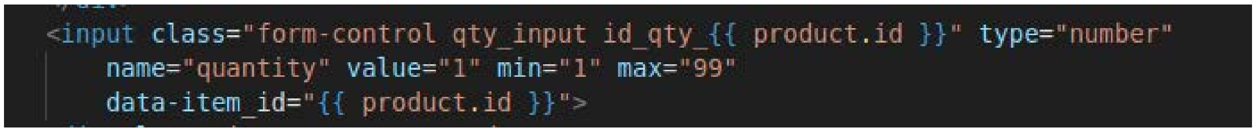
Now this took some time to investigate.

A while back, an issue had been spotted in the original Boutique Ado design: there are always two quantity forms in the HTML code - one for mobile screens, one for tablet and above - and only one is rendered at any time (the other one is hidden using Bootstrap's **d-none**), yet originally they both bore the same ID. The quantity input scripts, however, had been targetting the ID, resulting in only the *first* form (with the same ID) in HTML being targeted, this in turn causing the script to not work properly on screens larger than mobile. This was addressed by a brilliant fix by another Tutor back then.

"Maybe the fix had missed an edge case somehow", Peter thought.

But then he inspected the form fields in both forms, and found them substantially identical (in product_detail, the sizes were handled in a separate input field, so that was clearly not the issue here):

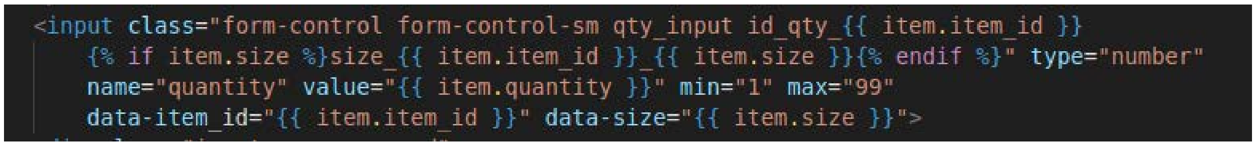
product_detail.html



```
<input class="form-control qty_input id_qty_{{ product.id }}" type="number"
    name="quantity" value="1" min="1" max="99"
    data-item_id="{{ product.id }}">
```

---

bag/quantity_form.html



```
<input class="form-control form-control-sm qty_input id_qty_{{ item.item_id }}
    {% if item.size %}size_{{ item.item_id }}_{{ item.size }}{% endif %}" type="number"
    name="quantity" value="{{ item.quantity }}" min="1" max="99"
    data-item_id="{{ item.item_id }}" data-size="{{ item.size }}">
```

So, the relevant classes matched, the min/max settings matched, the **`data-item_id`** values matched... Yet the two forms were behaving differently.

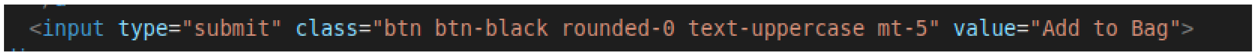
But why? How was that possible?

Peter had actually already had a theory at the very beginning, but had initially dismissed it as implausible.

However, having now exhausted all other possible explanations, he decided to pursue that suspicion once again.

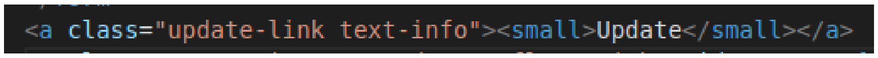
The only ever discernible and significant difference between the two forms was this:

The product_detail form was being submitted directly - using the **Add to Bag** button for the single product on the detail page

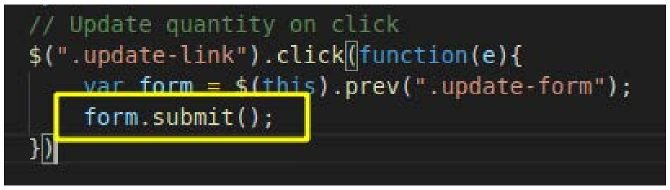


```
<input type="submit" class="btn btn-black rounded-0 text-uppercase mt-5" value="Add to Bag">
```

- whereas the **Update** link in bag.html was actually triggering a jQuery script that then submitted the form of that particular product in the bag, using `form.submit()`:



```
<a class="update-link text-info"><small>Update</small></a>
```

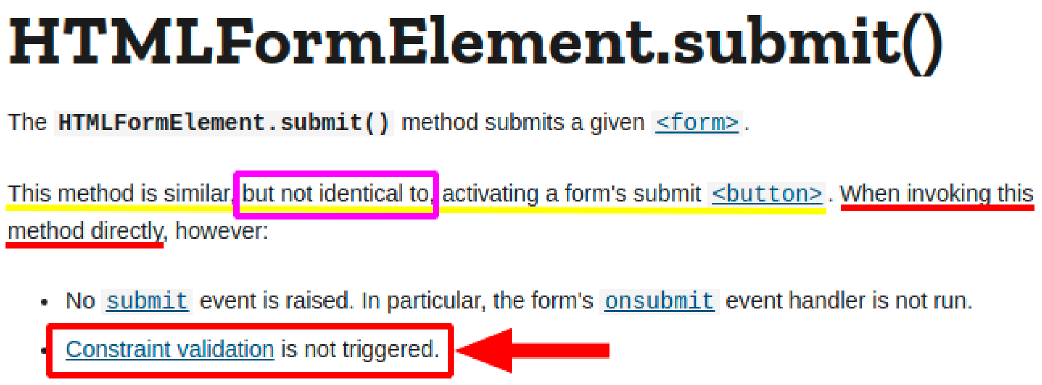


```
// Update quantity on click
$(".update-link").click(function(e){
    var form = $(this).prev(".update-form");
    form.submit();
})
```

“Fine, great find there”, ironised Peter, “but why would that make any difference?”

Shouldn’t **`form.submit()`** be the equivalent of actually submitting the form directly?

Well, to find that out, Peter then went to MDN Web Docs - specifically, to the form.submit() specification page - and, to his absolute astonishment, this is what he read:



HTMLFormElement.submit()

The `HTMLFormElement.submit()` method submits a given `<form>`.

This method is similar, but not identical to, activating a form's submit `<button>`. When invoking this method directly, however:

- No `submit` event is raised. In particular, the form's `onsubmit` event handler is not run.
- Constraint validation is not triggered.

This was one of those moments when one feels that not a light bulb but a massive spotlight had just been turned on above their head.

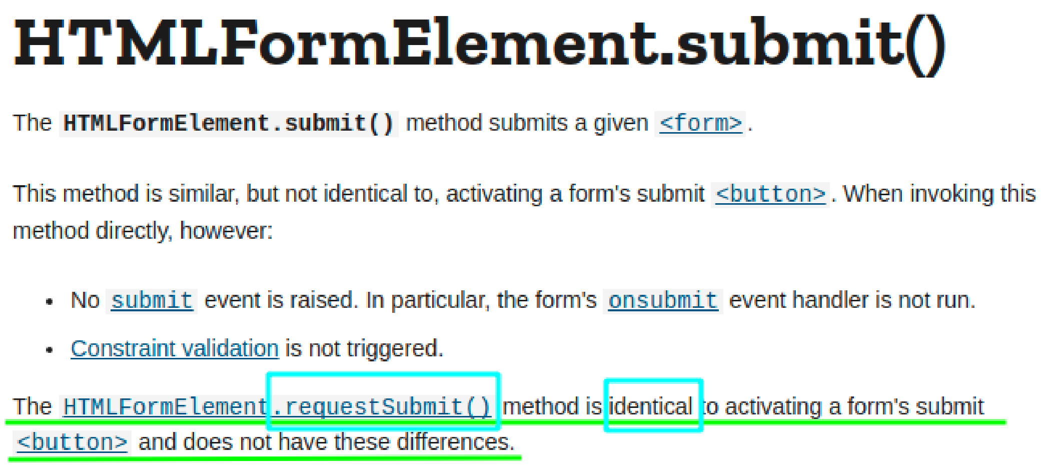
That was it… The fact that the bag form was being submitted via jQuery’s **`form.submit()`** was precisely the source of the different behaviour, and of the end issue - the quantity form field was not being checked for the input value constraint.

Peter was at the same time thrilled at the find, and a bit ashamed: more than two years into programming and more than a year into Tutoring, he had only just now learned this massive lesson about one of the most basic and frequent JS DOM manipulation methods.

But something else actually bothered him…

“If **`form.submit()`** doesn’t check for constraints, then can we even fix this?“

Luckily, MDN Web Docs answered that question immediately below:



HTMLFormElement.submit()

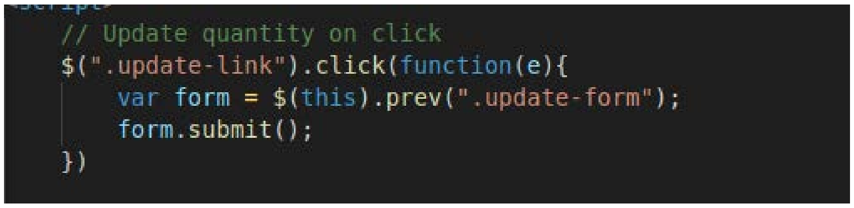
The `HTMLFormElement.submit()` method submits a given `<form>`.

This method is similar, but not identical to, activating a form's submit `<button>`. When invoking this method directly, however:

- No `submit` event is raised. In particular, the form's `onsubmit` event handler is not run.
- Constraint validation is not triggered.

The `HTMLFormElement.requestSubmit()` method is identical to activating a form's submit `<button>` and does not have these differences.

So, Peter went back to the code to test this.

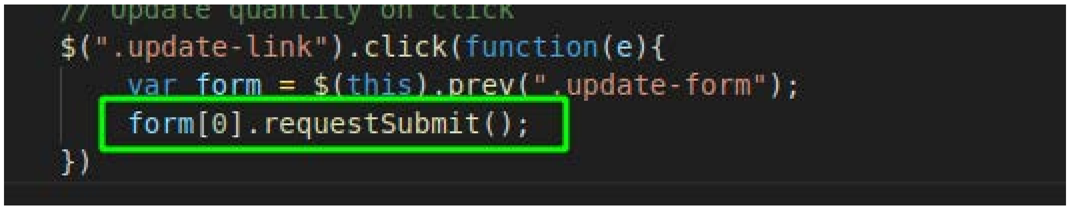


```
    // Update quantity on click
    $(".update-link").click(function(e){
        var form = $(this).prev(".update-form");
        form.submit();
    })
```

He realised that that **`form`** variable was actually a jQuery object (not a plain HTML element), so just attaching a Vanilla JS method to it wouldn't work.

The jQuery object needs to be indexed to get the actual HTML element out of it, that we can then attach the Vanilla JS method to.

So:

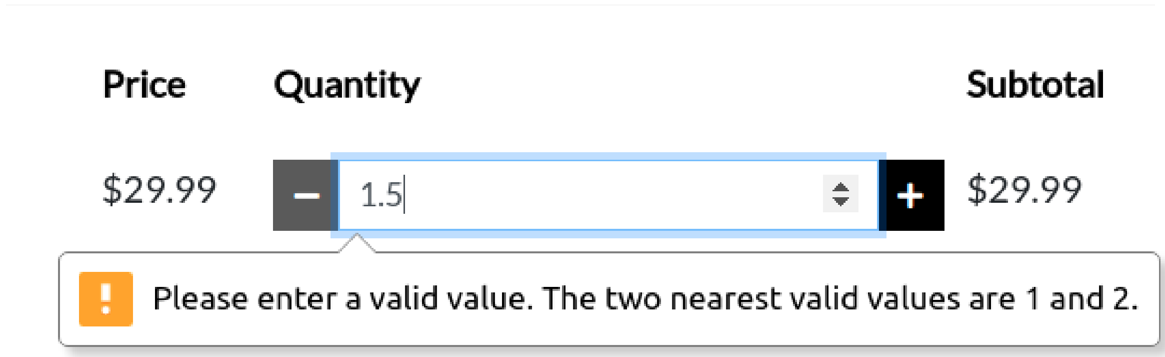


```
    // Update quantity on click
    $(".update-link").click(function(e){
        var form = $(this).prev(".update-form");
        form[0].requestSubmit();
    })
```

Thrilled, Peter ran the server, opened the app, and went to the bag (there was already an item in there).

He input **1.5** into the quantity field using the keyboard and, with almost childish excitement, clicked the **Update** button.

The result:



Price | Quantity | Subtotal
$29.99 | 1.5 | $29.99

Please enter a valid value. The two nearest valid values are 1 and 2.

Had anyone (but his dog) been able to see Peter's face at that moment, they would have seen an ear-to-ear smile... A smile that radiated joy, relief, triumph, and a little bit of healthy pride, all at the same time.

He immediately went to the student's conversation in the Tutor Tab and wrote a long message listing the whole investigation process, the finding, and the solution.

The student was immensely grateful, and happy.

But not nearly as happy as Peter. Not only had Peter successfully found the source of the issue and the explanation for it, he also managed to find and implement a solution and, on top of it all, he learned something new and very valuable for any future form scenario.

If that is not a perfect day in the life of a programmer, then I honestly don't know what is.

* * * * * * *

I hope you enjoyed this little detective story. 🙂

Remember that **Tutor Support is available 24h on weekdays and 9am-5pm (Irish time) on weekends**, so please don't hesitate to contact us if you need any assistance with a course lesson, challenge, project, or anything else content-related.

Happy coding! 🙂