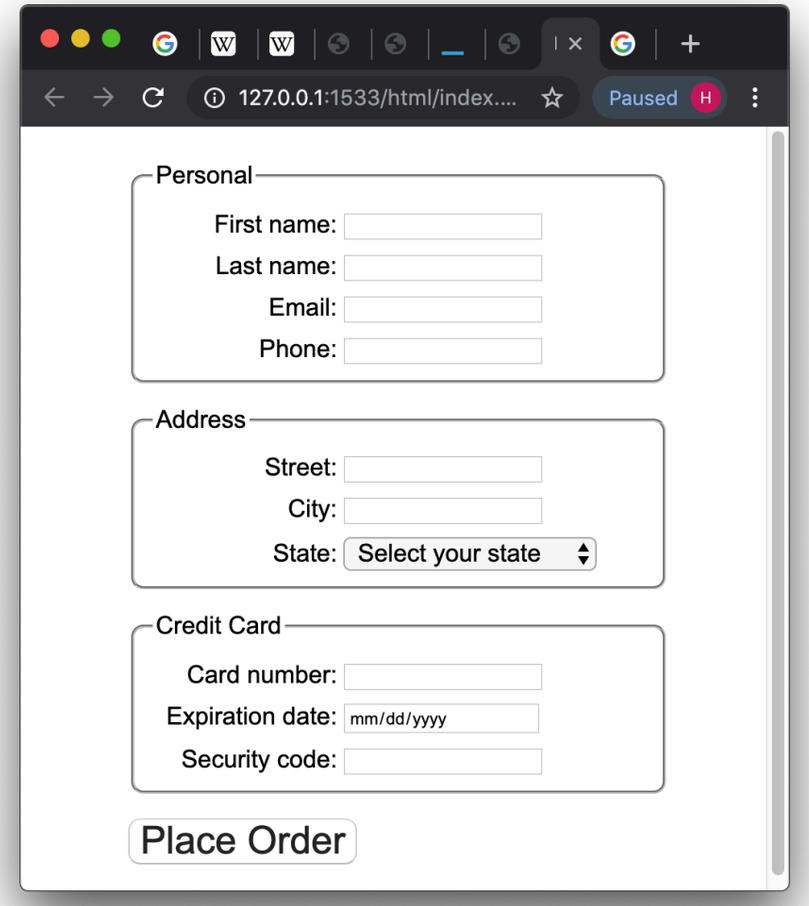
Rev. 08Aug2019

Exercise 51

WORKING WITH FORMS

Very often when we react with users, we do so through HTML forms.



Forms were an early innovation in HTML. As they were originally conceived, the developer would specify an `action` attribute to the `form` element. This attribute was the URL of a separate page. When the user submitted the form, all of the form information would be sent to the `action` URL for processing.

The trend for some time has been away from this default behavior. Instead, we write *single page applications* (SPAs) where the user stays on the same page and the content seen is swapped in and out.

Building complete SPAs using the extremely popular technologies, React and Redux, is covered in my CareerChanger course at codingcareer.com.

SPAs offer a much better user experience than did the old-style development, but it requires more work from you, the developer. One thing we'll need to do is to prevent the form's default behavior of a page load when a form is submitted. That's very simple:

```
const submitForm = event => {
  event.preventDefault();
}
document.querySelector('#theForm').addEventListener('submit', submitForm)
```

The use of `event.preventDefault()` stops the form's default behavior.

FORM VALIDATION

Speaking of old-style development, it used to be the case that the developer had to create their own mechanism for *form validation*. What is form validation and why do we need it? Suppose you have a form that asks the user for their email. The user, unwilling to give their email, skips over this form field. Or they type in `noneofyourbusiness`.

Form validation provides a line of defense against bad form input. As I said, it used to fall solely on the developer to create their own mechanism for this. But a later addition to HTML today offers the developer a great deal of help by means of built-in validation.

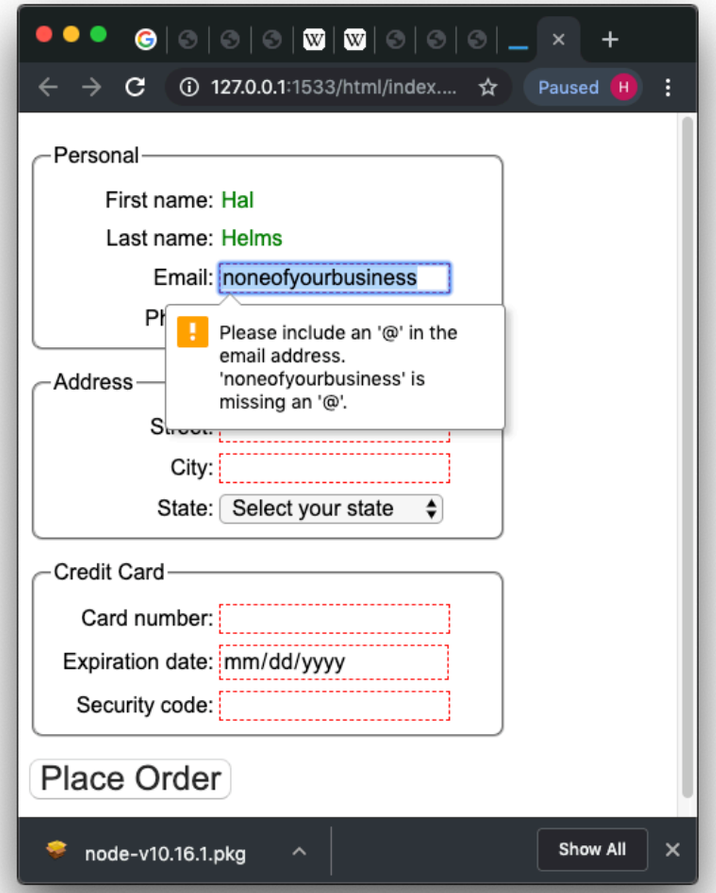
Adding `required` to any form field will ensure that the user types in *something*.

```
<input type="text" id="email" required />
```

But we can do even better. If we change the `type` from `text` to `email`, the browser will check for us whether what's entered at least matches the *pattern* for valid emails (although it can't ensure the email entered is real).

```
<input type="email" id="email" required />
```

Now, if the user types in something like `noneofyourbusiness`, they'll see instructions on the required way to fill out the text.



Built-in HTML validation can accommodate a variety of rules:

- `required`: some value is required in the field
- `minlength`: the provided value must be at least a certain number of characters in length
- `maxlength`: the provided value cannot exceed a certain number of characters in length
- `email`: the provided value must match the pattern for an email
- `tel`: the provided value must match the pattern for a phone number
- `date`: the provided value must be a date (when so defined, the form presents the user with a date picker)
- `pattern`: the provided value must match the pattern defined by a regex
- `min`: the provided numeric value must not be lower than specified

credit card. HTML doesn't have a built-in `type="credit-card"` option. For this, we will have to write a custom form validation. Luckily, it's pretty easy.

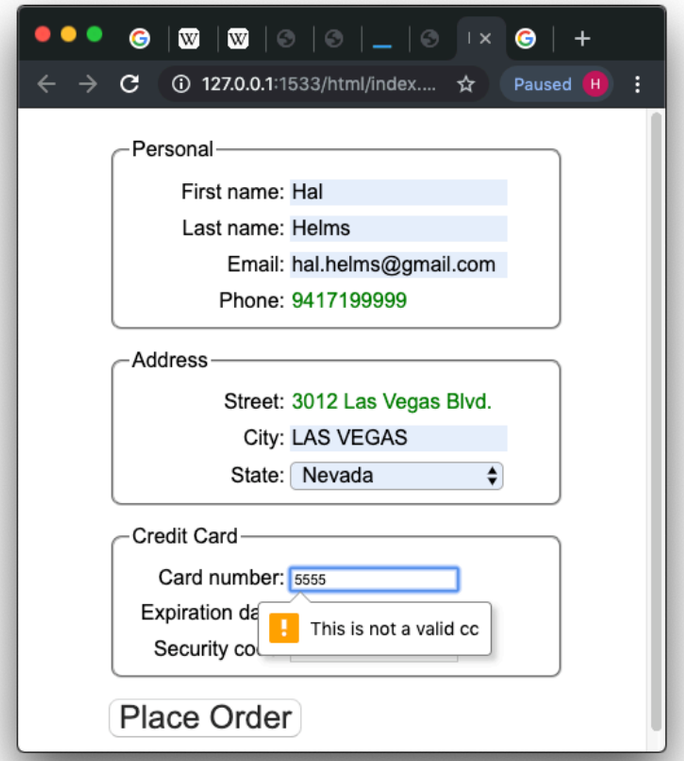
Let's start by creating a `checkCC` function that we'll use in conjunction with an event listener.

```
let checkCC = () => {
  let el = document.getElementById('cardNumber');
  let isValid = validCreditCard(el.value);
  console.log(isValid)
  if (isValid) {
    el.setCustomValidity('')
  } else {
    el.setCustomValidity('This is not a valid cc')
  }
}
```

I'm calling `validCreditCard`, passing it the card number entered by the user. By the looks of the rest of the code, that function returns a true/false value — but how does it make the determination on whether a credit card number is valid?

Happily, there's an algorithm for that, called Luhn's algorithm. And, like any good developer, I googled "Luhn algorithm", found an implementation that someone kindly posted, and will be using that. I placed it in a separate file, `validate-credit-card.js`, and included that in `index.html` so that my `index.js` has access to that.

There is some magic involved. There is a call to the form field's `setCustomValidity` function. Passing an empty string to this function tells the browser that validation has passed successfully. Passing anything else tells the browser that validation has *not* passed and the browser will use the text passed to it to alert the user of the situation.



And now, let's apply that to the form:

```
document.getElementById('cardNumber').addEventListener('change', checkCC)
```

You can use this technique for any type of custom form validation you need.