Creating, Styling, and Validating Web Forms

with examples and
hands-on exercises

WEBUCATOR

Version: 1.5.0

The Authors

Nat Dunn

Nat Dunn is the founder of Webucator (www.webucator.com), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at @natdunn and Webucator at @webucator.

Brian Hoke (Editor)

Brian Hoke joined Webucator as CEO in January, 2022. Brian Hoke was formerly Principal of Bentley Hoke, a web consultancy formed in Syracuse, New York, in 2000. The firm served the professional services, education, government, nonprofit, and retail sectors with a variety of development, design, and marketing services. Previously, Brian served as Director of Technology, Chair of the Computer and Information Science Department, and Dean of Students at Manlius Pebble Hill School, an independent day school in DeWitt, NY. Before that, Brian taught at Insitut auf dem Rosenberg, an international boarding school in St. Gallen, Switzerland. Brian holds degrees from Hamilton and Dartmouth colleges.

Class Files

Download the class files used in this manual at https://static.webucator.com/media/public/materials/classfiles/WFM101-1.5.0-creating-styling-and-validating-web-forms.zip.

Errata

Corrections to errors in the manual can be found at https://www.webucator.com/books/errata/.

Table of Contents

LESSON 1
HTML Forms

Topics Covered

☑ How HTML forms work.

☑ The post and get methods.

☑ Form elements.

☑ Labels.

Introduction

In this lesson, you will learn to work with HTML forms.

1.1. How HTML Forms Work

HTML forms are used for submitting data back to a script on the server for data processing. When a form is submitted, the data in the form fields is passed to the server as name-value pairs. Server-side scripts, which can be written in several different languages, are used to process the incoming data and return a new HTML page to the browser. The page returned to the browser could be anything from a "Thank you for registering" message to a list of search results generated from a database query.

The form processing occurs in the following sequence:

1. The user fills out the form and submits the form data using a "submit" button.
2. The data is sent to the web server.

3. A script on the web server processes the form, possibly interacting with the file system, one or more databases, a mail server, or any number of other applications.
4. The script generates an HTML page, which the server returns to the client for display.

1.2. The `form` Element

HTML forms are created using the `<form>` tag, which takes two main attributes: `action` and `method`.

The `action` specifies the URL of the page that processes the form. The `method` attribute has two possible values: `post` and `get`. Here is an example of a `form` element:

```
<form method="post" action="process-form.cfm">
  <!--form fields go here-->
</form>
```

❖ 1.2.1. Get vs. Post

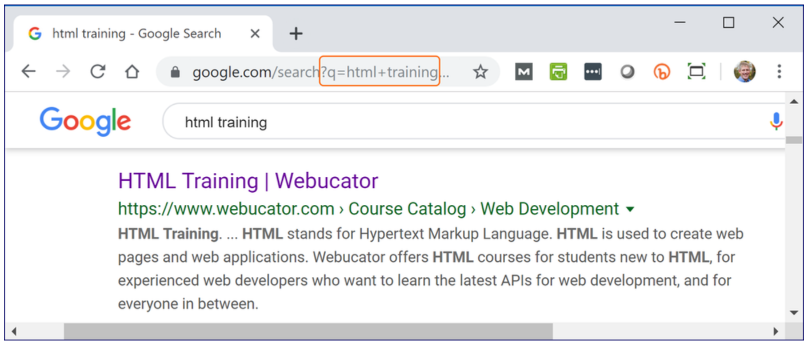
The value of the `method` attribute determines how the form data will be passed to the server.

get

When using the `get` method, which is the default, form data is sent to the server in the URL as a *query string*. The query string is appended to the website address starting with a question mark (?) and followed by name-value pairs delimited (separated) by an ampersand (&). A URL with a query string might look like this:

```
https://www.example.com?firstname=Nat&lastname=Dunn
```

The `get` method is commonly used by search engines, because it allows the resulting page to be bookmarked. For example, Google uses the `get` method. You can tell by looking at the location bar after doing a search:


html training - Google Search
google.com/search?q=html+training...
Google
html training

HTML Training | Webucator
https://www.webucator.com › Course Catalog › Web Development
HTML Training. ... **HTML** stands for Hypertext Markup Language. **HTML** is used to create web pages and web applications. Webucator offers **HTML** courses for students new to **HTML**, for experienced web developers who want to learn the latest APIs for web development, and for everyone in between.

post

When `post` is used, the name-value pairs are not sent as part of the query string. Instead, they are sent behind the scenes. This has the advantage of keeping the values hidden from anyone looking over the user's shoulder. Two other advantages of the `post` method are:

1. It allows for much more data to be submitted (i.e., larger forms).
2. It allows for files to be uploaded to the server.[1]

Use Post for Most Forms

As a general rule, you should use `post` unless you want the user to be able to bookmark or share (e.g., via email) the resulting web page.

1.3. Form Elements

This section describes the different form elements that can be used to input data into a form. As you will see, many of these elements, but not all, are created with the `<input>` tag.

1. Files can be uploaded to the server via the `file` input type. The tag syntax is: `<input type="file" name="filename">`.

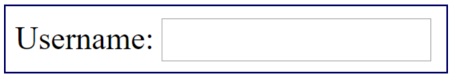
❖ 1.3.1. id and name Attributes

Form fields (also called controls) take both the `name` attribute and the `id` attribute. They are used for different purposes:

- The `name` attribute is used to hold the value of the field when data is sent to the server.
- The `id` attribute is used by the browser to identify a specific element.

❖ 1.3.2. Text Fields

Text fields are created with the `<input>` tag with the `type` attribute set to "text". They are used for single lines of text:



The code to create the `input` element shown above is:

```
Username: <input type="text" name="username" id="username">
```

As `text` is the default `type` for `input` elements, if the `type` attribute is absent, the `input type` will be `text`. So, the above code can also be written:

```
Username: <input name="username" id="username">
```

❖ 1.3.3. Labels

Form element labels should be put in `<label>` tags. Labels can be associated with form elements using two methods:

1. Using the `for` attribute of the `<label>` to point to the `id` attribute of the form element.
2. Wrapping the form element in the `<label>` tag.

```
Method 1
<label for="username">Username:</label>
<input type="text" name="username" id="username">
```

```
Method 2
<label>
  Username:
  <input type="text" name="username" id="username">
</label>
```

We will mostly use the first method.

❖ 1.3.4. Text-like Input Types

There are many `input` types that are similar to the `text` type:

1. `password`
2. `date`
3. `time`
4. `datetime-local`
5. `month`
6. `week`
7. `color`
8. `email`
9. `tel`
10. `url`
11. `search`
12. `number`

Not all of these types are supported by all browsers. When a browser does not support a certain type, it will fall back to using a standard `text` type, so you can technically use all of these types today. However, **we recommend not using the `date` and `time` types until there is more consistent browser support**. We will explain why soon.

Common Attributes for Text-like Inputs

1. `type` – Input type (e.g., `text`, `tel`, etc.). Applies to all text-like inputs.
2. `name` – Variable name used to send data to server. Applies to all text-like inputs.

3. `id` – Variable name used to identify field in the browser. Applies to all text-like inputs.
4. `value` – Initial value in the field. Applies to all text-like inputs.
5. `size` – Approximate number of characters visible in the field. Applies to `text`, `search`, `tel`, `url`, `email`, and `password`.
6. `minlength` – Minimum number of characters that must be entered. Applies to `text`, `search`, `tel`, `url`, `email`, and `password`.
7. `maxlength` – Maximum number of characters that can be entered. Applies to `text`, `search`, `tel`, `url`, `email`, and `password`.
8. `placeholder` – A hint indicating what should be entered in the field. Applies to all text-like inputs.
9. `pattern` – A regular expression expressing a valid value for the field. Applies to `text`, `search`, `tel`, `url`, `email`, and `password`.
10. `required` – When present, the user must fill in a value before submitting the form. Applies to all text-like inputs.
11. `autofocus` – Instructs the browser to place focus on that field allowing the user to begin typing as soon as the page loads. Applies to all text-like inputs.
12. `autocomplete` – Used to override the browser's or form element's autocomplete behavior on a field-by-field basis. When used, it is usually set to "off".[2] Applies to all text-like inputs, except `password`.

Be Careful with Autofocusing

Autofocusing on a form element can cause problems for people using screen readers. For sighted people, it's generally okay if we provide one focus point for the keyboard (i.e., `autofocus`) and another one for the eyes (e.g., instructions for filling out the form), but for people using screen readers, there is only one focus point. So be careful not to skip over important contextual content when directing focus to a form field using `autofocus`.

❖ 1.3.5. placeholder Attribute

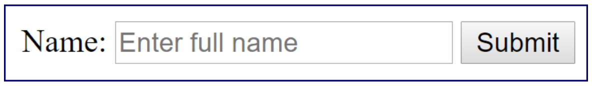
The `placeholder` attribute is used to add placeholder text to the form field. The placeholder text will go away as soon as the user begins typing in the field. The following code illustrates:

2. See `https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#autofill`.

Demo 1.1: Forms/Demos/placeholder.html

```
-------Lines 1 through 9 Omitted-------
  <label for="fullname">Name: </label>
  <input type="text" name="fullname" id="fullname"
    placeholder="Enter full name">
  <input type="submit">
-------Lines 14 through 16 Omitted-------
```

Here's what it looks like in the browser:



❖ 1.3.6. pattern Attribute

The `pattern` attribute is used to force a specific pattern (via a regular expression[3]) within a form field. You can use a `placeholder` to give an example of a valid entry:

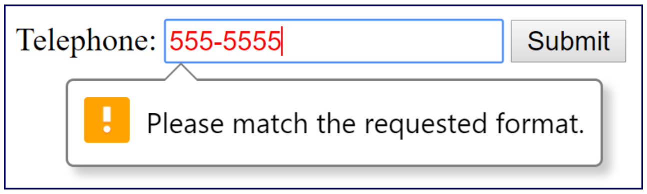
Demo 1.2: Forms/Demos/pattern.html

```
-------Lines 1 through 14 Omitted-------
  <label for="telephone">Telephone: </label>
  <input type="tel" name="telephone" id="telephone"
    placeholder="(555) 555-5555"
    pattern="^\(?([2-9]\d\d)\)?[\-\. ]?([2-9]\d\d)[\-\. ]?(\d{4})$">
-------Lines 19 through 22 Omitted-------
```

Open this file in the browser and enter data in the field. It should remain red[4] until the value is a valid 10-digit U.S.-style phone number. It will allow for parentheses around the area code and for dashes, spaces, and dots as separators. It will also allow for no separators. Note the first digit may not be a 0 or a 1.

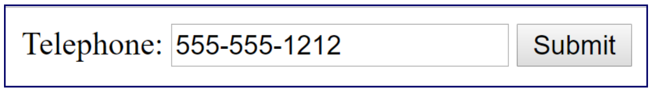
When the user tries to submit with an invalid pattern, an error will appear:

3. Regular expressions are used by many programming languages for pattern matching. The syntax is complex, but super powerful.
4. Note that we use CSS to make the invalid content red. By default, you won't know it's invalid until you submit.



Please match the requested format.

When the user enters a valid pattern, the field will indicate that it is valid (e.g., by changing the font color from red to black):

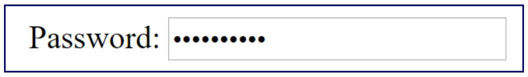


❖ 1.3.7. Password Fields

Password fields are similar to text fields. They are coded as follows:

```
<label for="pw">Password:</label>
<input type="password" name="pw" id="pw" size="10" maxlength="12">
```

The only difference between a password field and a text field is that the value entered in a password field is disguised so that onlookers cannot see it:



Not Really Secure

Note that there is no additional security provided by password fields beyond obscuring the text you enter in the field. Passwords are not sent to the server any differently than other fields.

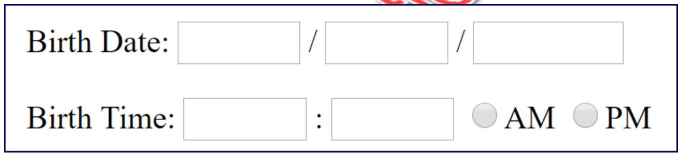
❖ 1.3.8. Date and Time Fields

The `date`, `time` and `datetime-local` field controls in Google Chrome are beautiful, both in their presentation and in their control of the data entry. When these controls are used, you can count on the data being in the formats shown in the following table:

Date and Time Data Formats

Input Type	Data Format	Example(s)
`date`	`yyyy-mm-dd`	`1991-06-12`
`time`	`hh:mm`	`03:05`, `15:05`
`datetime-local`	`yyyy-mm-ddThh:mm`	`1991-06-12T03:05`
`month`	`yyyy-mm`	`1776-07`
`week`	`yyyy-Www`	`2028-W02`

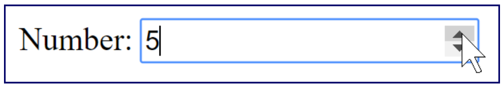
Knowing the format of incoming data makes it much easier to process the data on the server side. Unfortunately, users of browsers (most notably Safari) that do not support these field types are presented with basic text fields instead of the more modern date and time controls. Such users are not likely to enter data in the formats shown in the table above. That means that you have to write code to validate the user-entered data. While this is possible, it would still be confusing to the user. As such, for now, you are better off taking other approaches to collecting date and time values. One simple approach is to use a combination of `number` inputs and `radio` buttons. We will learn about those soon. In the meantime, here is a screenshot of what the form might look like in the browser:



While this isn't as pretty as the date and time controls used by some browsers, for now, it provides a safer way of collecting accurate data from the user.

❖ 1.3.9. Number Fields

Browsers present up and down buttons (*spinboxes*) to scroll through numbers, and they also allow you to use the up and down arrows on the keyboard:



The step Attribute

By default, numbers increment by `1` and any non-integer (e.g., a decimal like `0.5`) is considered invalid. However, you can change the increment using the `step` attribute:

```
<input id="amount" name="amount" type="number" step=".01">
```

With `step` set to ".01", valid numbers can have a decimal point followed by one or two digits.

The min and max Attributes

You can control the range of possible values using the `min` and `max` attributes:

```
<input id="amount" name="amount" type="number" step=".01" min="0" max="100">
```

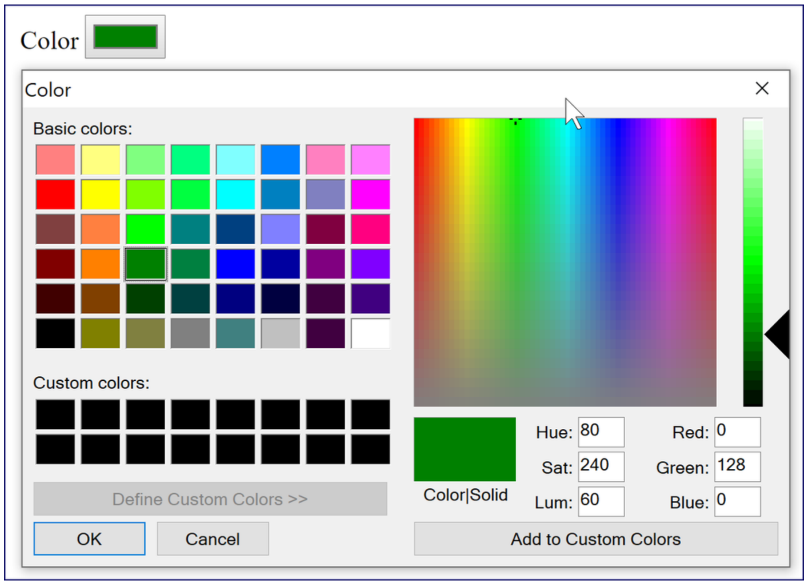
With the code above, numbers below `0` and above `100` will be invalid.

❖ 1.3.10. Color Fields

Most browsers will present a color picker when the user focuses on a `color` field:

```
<input id="color" name="color" type="color">
```

This will appear different in different browsers. The following screenshot shows how it appears in Google Chrome:



Hue: 80
Sat: 240
Lum: 60
Red: 0
Green: 128
Blue: 0

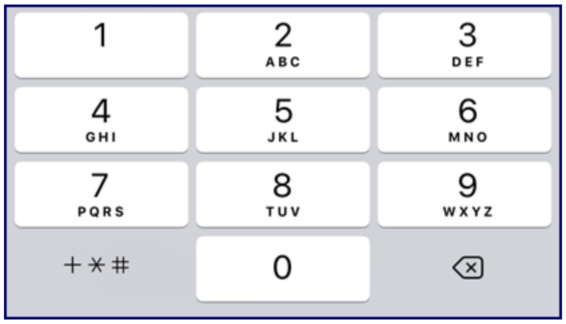
❖ 1.3.11. Tel, URL, and Email Fields

tel

```
<input id="telephone" name="telephone" type="tel">
```

On desktop browsers, you don't really gain anything by using the `tel` input type. As telephone numbers can come in all different formats, there are no constraints on what can be entered here. You could, however, add your own custom validation to all telephone inputs using the `pattern` attribute as demonstrated earlier.

Also, user agents are free to provide a different/better means for filling out input fields based on their type. For example, the iPhone provides a more appropriate interface (presenting the user with the phone keypad) for filling out fields of the `tel` type:



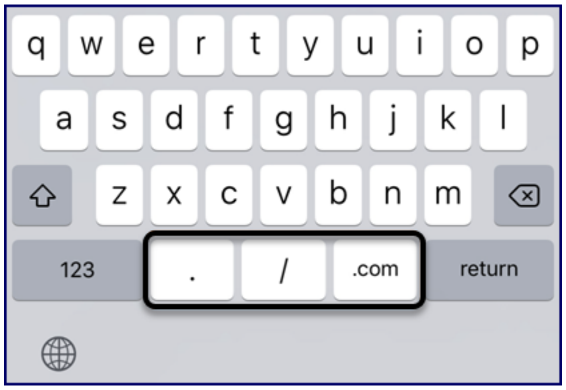
url and email

```
<input id="url" name="url" type="url">
<input id="email" name="email" type="email">
```

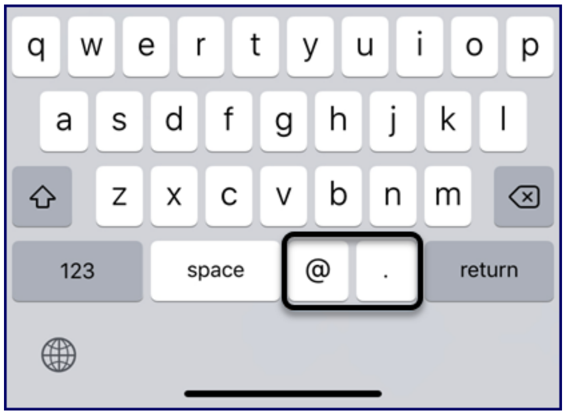
Browsers provide validation for `url` and `email` fields to make sure the user enters valid data.

Also, as with `type="tel"`, user agents are free to provide a different/better means for entering URLs and email addresses.

For example, for `url` types, the iPhone provides keys for **.**, **/** and **.com** and does not provide a **Space** key, as spaces are not allowed in URLs:



For emails, the iPhone provides **@** and **.** keys:

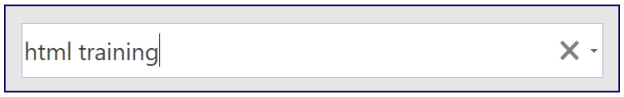


Interestingly enough, the iPhone also provides a **"space"** key for emails. This is because `email input` types can include a `multiple` attribute, which, when included, allows users to enter multiple emails delimited by spaces. If the iPhone were a little smarter, it would only include the **Space** key when the `multiple` attribute was present.

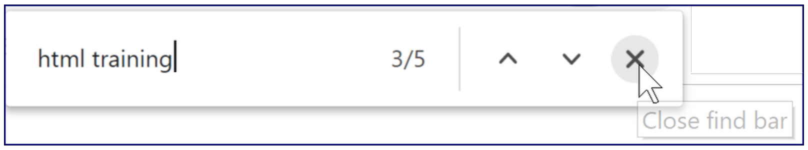
❖ 1.3.12. Search Fields

```
<input id="search" name="search" type="search">
```

Most `input` fields are meant to be filled out only one time and then submitted for processing. But a search box is a bit different. For example, consider Microsoft Word's search box:



Notice the **x** used for clearing the box. If you look at search boxes in other applications, you'll notice many of them also provide a simple way to clear the text. Modern web browsers take the same approach that Word does. Here's Chrome's search box:



Note that the **x** doesn't show up until you have entered some text into the field.

❖ 1.3.13. Hidden Fields

Hidden fields are created with the `input` element with the `type` attribute set to `hidden`. They are used to pass name-value pairs to the server without displaying them to the user. Hidden fields are often used to identify the product being ordered on an e-commerce page. For example:

```
<input type="hidden" name="product-id" id="product-id" value="42">
```

Beware of Hackers

Although the user can't change the value of an input field via the form, savvy users can change the value of any field, including `input` fields, using the browser's developer tools, so you must always include some sort of server-side validation, meaning you must have code on the server that verifies that the data coming in from the form is valid and safe.

1.4. Buttons

Submit and reset buttons can both be created with the `<input>` tag.

❖ 1.4.1. Submit Button

A simple `submit` button looks like this:

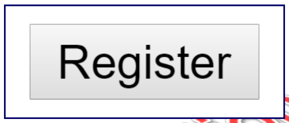
Sample code for a submit button:

```
<input type="submit" name="submitbtn" id="submitbtn">
```

Different browsers format buttons in different ways and some have different default text for submit buttons. Use the value attribute to explicitly set the text of the button:

```
<input type="submit" name="submitbtn" id="submitbtn" value="Register">
```

Now the button will appear as follows:



When a form has a submit button, it can be submitted either by clicking the button or by pressing the **Enter** key when an input element has focus.

When a submit button is clicked, the name and value of that button are sent to the server (as a name-value pair). This can be useful in the event that a form has multiple submit buttons as the processing page can be set to behave differently depending on which button is clicked to submit the form.

❖ 1.4.2. Reset Button

A reset button is used to set all the form fields back to their original values. A reset button looks like this:

Most browsers use "Reset" as the default text. While this can be changed with the value attribute, it is generally a better practice to leave the default value unchanged, or even to explicitly set it to "Reset" as users are likely familiar with a standard **Reset** button.

Sample code for a `reset` button:

```
<input type="reset" name="resetbtn" id="resetbtn" value="Reset">
```

❖ 1.4.3. Button Buttons

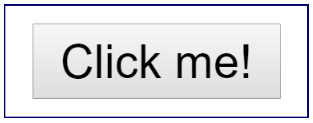
Buttons can also be created using the `<button>` tag with an optional `type` attribute, which defaults to “submit” if not present. Other possible values for `type` are “reset” and “button”.

`<button type="submit">` and `<button type="reset">` can be used interchangeably with `<input type="submit">` and `<input type="reset">`. The text between the opening and closing `<button>` tags shows up on the button.

Sample code for a `button` button:

```
<button type="button" id="mycustombtn">Click me!</button>
```

This will appear as follows:



Button Buttons and JavaScript

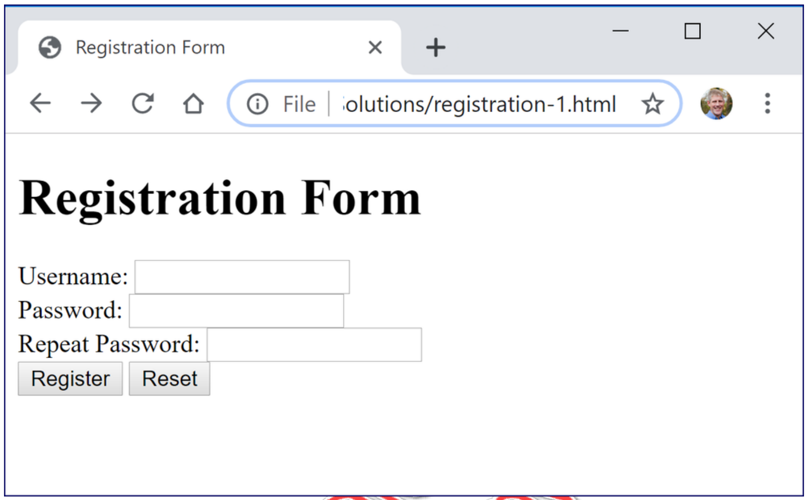
Button buttons are often used in conjunction with JavaScript to add custom behaviors to a web page. Open `forms/Demos/button-buttons.html` in your browser to see some fun things you can do with buttons and JavaScript.

Exercise 1: Creating a Registration Form

20 to 40 minutes

In this exercise, you will begin to create a registration form.

The form should appear as follows:



Registration Form

Username:
Password:
Repeat Password:
Register Reset

1. Open `Forms/Exercises/registration.html` for editing.
2. Add a `form` element to the page.
   - The `action` should be:

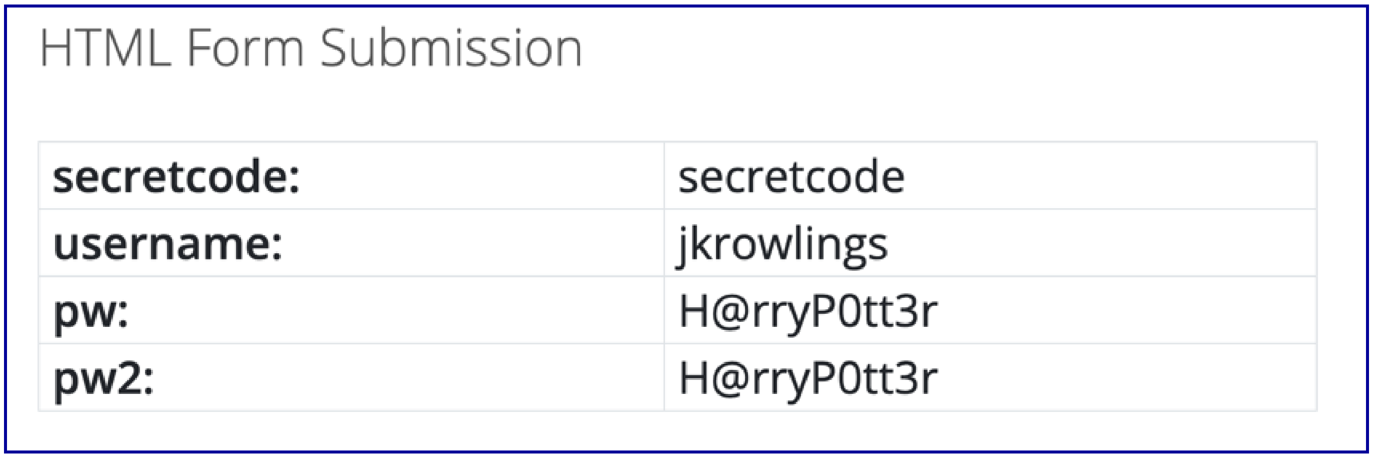
     ```
     https://www.webucator.com/materials/htm101/
     ```

   - The `method` should be "post".
3. Add the following form elements:
   - Hidden field: `name` and `id` should be "secretcode"; `value` should be "42".
   - Text field: `name` and `id` should be "username". The `input` should have a corresponding `label`.
   - Password field: `name` and `id` should be "pw". The `input` should have a corresponding `label`.
   - Repeat password field: `name` and `id` should be "pw2". The `input` should have a corresponding `label`.

- Submit button: `value` should be “Register”.
- Reset button.

4. When you are done, open the page in a browser and fill out and submit the form.

You will need internet access to see the resulting page, which will look something like this:



HTML Form Submission

secretcode:	secretcode
username:	jkrowlings
pw:	H@rryP0tt3r
pw2:	H@rryP0tt3r

Note that the resulting page shown above is not something you coded. It is the `action` page we used in the form. This page just dumps back the data submitted in the form. We use it simply to show you that the form does indeed get submitted and the data is now available on the server.

Solution: Forms/Solutions/registration-1.html

```html
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1.0">
<title>Registration Form</title>
</head>
<body>
<h1>Registration Form</h1>
<form method="post"
      action="https://www.webucator.com/materials/htm101/">
<input type="hidden" name="secretcode" id="secretcode" value="42">
<div>
  <label for="username">Username:</label>
  <input type="text" name="username" id="username" size="15">
</div>
<div>
  <label for="pw">Password:</label>
  <input type="password" name="pw" id="pw" size="15">
</div>
<div>
  <label for="pw2">Repeat Password:</label>
  <input type="password" name="pw2" id="pw2" size="15">
</div>
<div>
  <input type="submit" name="submitbutton" id="submitbutton"
    value="Register">
  <input type="reset" name="resetbutton" id="resetbutton">
</div>
</form>
</body>
</html>
```

Note that we have used `div` elements to group the `label-input` pairs.

Be sure to correct any mistakes that you made as you will be starting with your solution to this exercise in the next exercise.

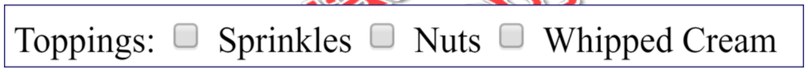
1.5. Checkboxes

Checkboxes are created with the `input` element with the `type` attribute set to "checkbox". The default value for a checkbox is "on". Although the value of a checkbox can be changed with the `value` attribute, there is usually no reason to do so, as the name-value pair only gets sent to the server if the checkbox is checked. In other words, the code on the server only needs to check for the existence of the variable name to see if the checkbox was checked or not.

Take a look at the following code:

```
Toppings:
<input type="checkbox" name="sprinkles" id="sprinkles">
<label for="sprinkles">Sprinkles</label>
<input type="checkbox" name="nuts" id="nuts">
<label for="nuts">Nuts</label>
<input type="checkbox" name="whip" id="whip">
<label for="whip">Whipped Cream</label>
```

This will appear as follows in the browser:



❖ 1.5.1. Checked By Default

Checkboxes can be set to be checked by default using the `checked` attribute:

```
<input type="checkbox" checked name="sprinkles" id="sprinkles">
```

❖ 1.5.2. Making a Checkbox Required

To force the user to check a checkbox (e.g., to accept a user agreement), checkboxes can be set to be required using the `required` attribute:

```
<input type="checkbox" required name="terms" id="terms">
<label for="terms">Check to indicate that you accept our terms.</label>
```

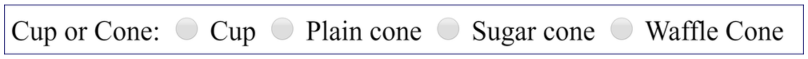
1.6. Radio Buttons

Radio buttons are created with the `input` element with the `type` attribute set to "radio". Radio buttons generally come in groups, in which each radio button has the same name. Only one radio button in the group can be checked at any given time. Each radio button in the group should have a unique value – the value to be sent to the server if that radio button is selected.

Take a look at the following code:

```
Cup or Cone?
<label>
  <input type="radio" name="container" value="cup">
  Cup
</label>
<label>
  <input type="radio" name="container" value="plaincone">
  Plain cone
</label>
<label>
  <input type="radio" name="container" value="sugarcone">
  Sugar cone
</label>
<label>
  <input type="radio" name="container" value="wafflecone">
  Waffle cone
</label>
```

This will appear as follows in the browser:


Cup or Cone: ○ Cup ○ Plain cone ○ Sugar cone ○ Waffle Cone

❖ 1.6.1. Radio Buttons, Labels, and the id Attribute

You will notice that we used `<label>` differently with radio buttons. Instead of using the `for` attribute, we wrapped each radio button in `<label>` tags. This is because our radio buttons don't include `id` attributes.

In form elements such as text fields and checkboxes, the `id` is usually the same as the name. Remember that `id`s and `name`s serve different purposes. Again, here is the difference between these attributes:

1. The `name` attribute is used to hold the value of the field when data is sent to the server.
2. The `id` attribute is used by the browser to identify a specific element.

In most cases, it's simplest to use the same value for both the `name` and the `id`. But for radio buttons, this isn't possible, because radio buttons in the same set must all have the same `name`, but they cannot all have the same `id`. All `id` values must be unique on a given page.

The following code shows radio buttons with `id`s:

```
Dominant Hand:
<input type="radio" name="dom-hand" id="hand-right" value="right">
<label for="hand-right">Right</label>
<input type="radio" name="dom-hand" id="hand-left" value="left">
<label for="hand-left">Left</label>
```

❖ 1.6.2. Checked By Default

Like checkboxes, radio buttons can be set to be checked by default using the `checked` attribute:

```
<input type="radio" name="container" value="cone" checked>
```

❖ 1.6.3. Requiring a Selection

To make a selection required, add the `required` attribute to at least one of the radio buttons in the named group:

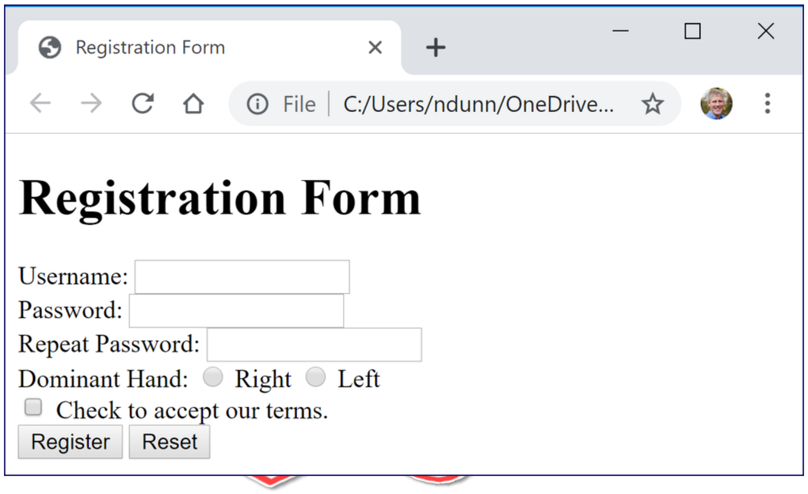
```
Dominant Hand:
<input type="radio" name="dom-hand" id="hand-right" value="right" required>
<label for="hand-right">Right</label>
<input type="radio" name="dom-hand" id="hand-left" value="left" required>
<label for="hand-left">Left</label>
```

There is no effective difference between adding the `required` attribute to one radio button or to all the radio buttons in a named group, but the code is clearer if you add it to all of them. Marking just one as required makes it look like you have to choose that option, which is not the case.

Exercise 2: Adding Checkboxes and Radio Buttons

10 to 15 minutes

In this exercise, you will add a checkbox and radio buttons to the registration form. On completion, the form should look like this:



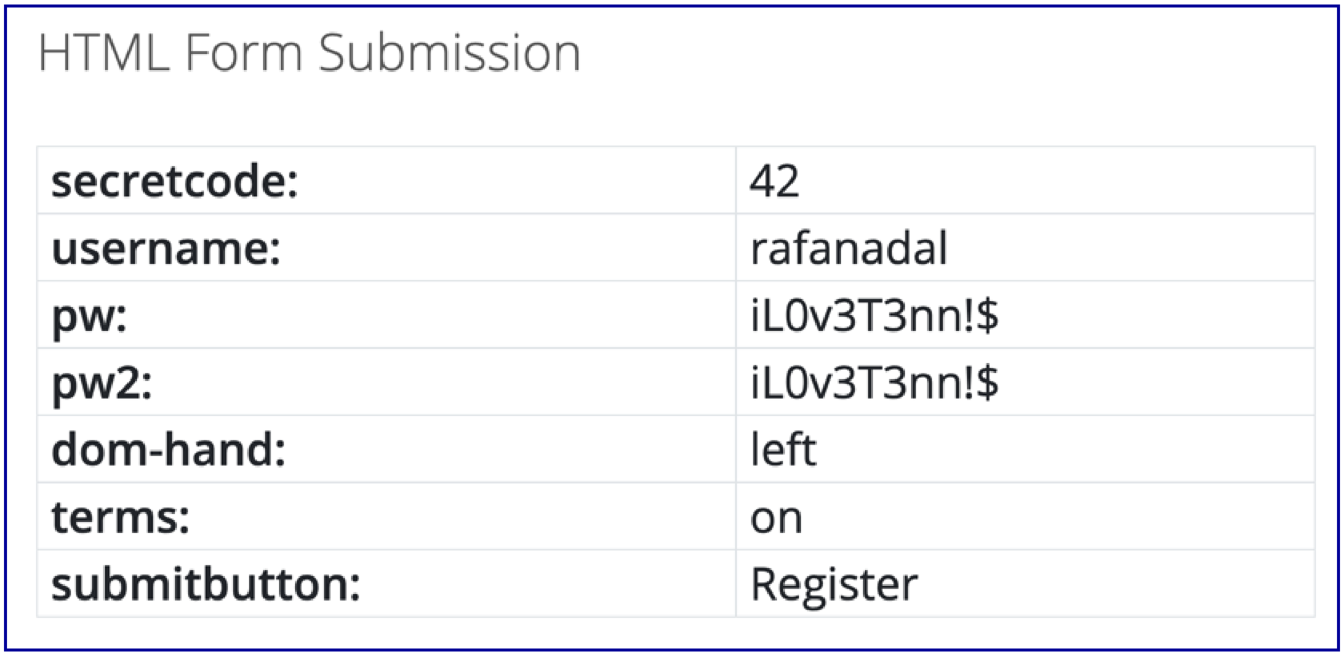
Registration Form

Username:
Password:
Repeat Password:
Dominant Hand: Right Left
Check to accept our terms.
Register Reset

1. Open `Forms/Exercises/registration.html` for editing if you don't have it open already.
2. Add the following `input` elements:
   A. Two radio buttons:
      i. The name should be "dom-hand" for both.
      ii. The `id` of each should be unique, say "hand-right" and "hand-left".
      iii. The `value` should also be unique, say "right" and "left".
      iv. The user should see the radio buttons labeled as "Right" and "Left", respectively.

   B. A checkbox: the `name` and `id` should both be "terms" and the field should be required.
3. When you are done, open the page in a browser and fill out and submit the form. You shouldn't be able to submit unless you have accepted the terms.

You will need internet access to see the resulting page, which will look something like this:



HTML Form Submission

secretcode:	42
username:	rafanadal
pw:	iL0v3T3nn!$
pw2:	iL0v3T3nn!$
dom-hand:	left
terms:	on
submitbutton:	Register

Solution: Forms/Solutions/registration-2.html

```
-------Lines 1 through 24 Omitted-------
<div>
  Dominant Hand:
  <label>
    <input type="radio" name="dom-hand" id="hand-right" value="right">
    Right
  </label>
  <label>
    <input type="radio" name="dom-hand" id="hand-left" value="left">
    Left
  </label>
</div>
<div>
  <input type="checkbox" name="terms" id="terms" required>
  <label for="terms">Check to accept our terms.</label>
</div>
-------Lines 40 through 47 Omitted-------
```

Be sure to correct any mistakes that you made as you will be starting with your solution to this exercise in the next exercise.

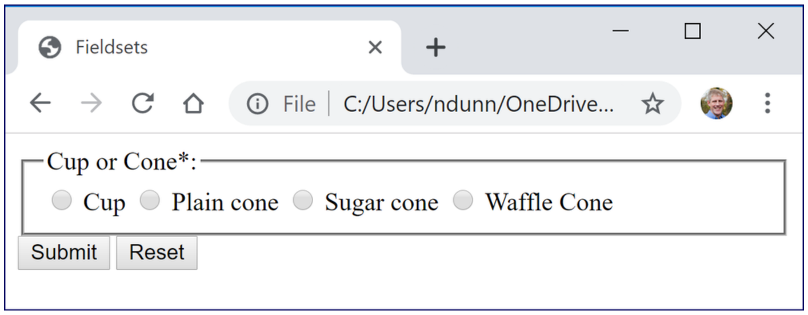
1.7. Fieldsets

The `fieldset` element is used to group a set of inputs. The `legend` element is used within a `fieldset` to give the group of elements a legend, which is similar to a caption. Here is a simple example with radio buttons:

Demo 1.3: Forms/Demos/fieldset.html

```
-------Lines 1 through 7 Omitted-------
<body>
<form method="post"
      action="https://www.webucator.com/materials/htm101/">
  <fieldset>
    <legend>Cup or Cone*:</legend>
    <label>
      <input type="radio" name="container" value="cup" required>
      Cup
    </label>
    <label>
      <input type="radio" name="container" value="plaincone"
        required>
      Plain cone
    </label>
    <label>
      <input type="radio" name="container" value="sugarcone"
        required>
      Sugar cone
    </label>
    <label>
      <input type="radio" name="container" value="wafflecone"
        required>
      Waffle Cone
    </label>
  </fieldset>
  <input name="submitbutton" id="submitbutton" type="submit">
  <input name="resetbutton" id="resetbutton" type="reset">
</form>
-------Lines 36 through 37 Omitted-------
```

And here is how it appears in Google Chrome:



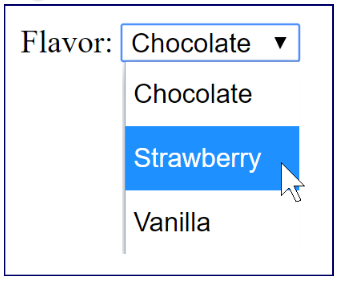
1.8. Select Menus

Select menus are created with the `select` element, which must contain one or more `option` elements. The text between the opening and closing `<option>` tags appears in the select menu. The value of the `option` element's `value` attribute is what gets passed to the server if that `option` is selected.

Here is a simple example:

```
<label for="flavor">Flavor:</label>
<select name="flavor" id="flavor">
  <option value="chocolate">Chocolate</option>
  <option value="strawberry">Strawberry</option>
  <option value="vanilla">Vanilla</option>
</select>
```

And here is how it appears in Google Chrome:



The `<select>` and `<option>` tags' attributes are listed below:

select Attributes

1. `size` – Number of options to appear at once.
2. `multiple` – Indicates that multiple options can be selected. Value must be "multiple" or left out.
3. `required` – When present, the user must select an option that has a value at least one character long.

option Attributes

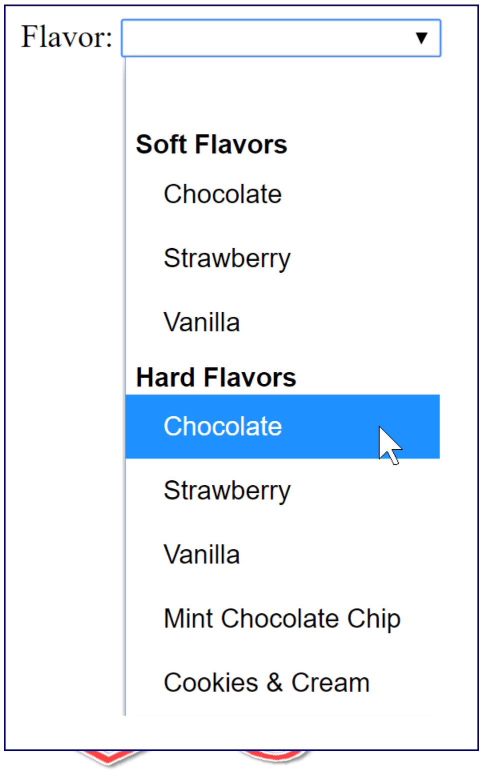
1. `value` – Value to send to server if option is selected.
2. `selected` – Indicates that option is pre-selected. Value must be "selected" or left out.

❖ 1.8.1. Option Groups

Options can be arranged in groups using the `<optgroup>` tag. The `label` attribute is used to set the option group heading.

```
<label for="flavor">Flavor:</label>
<select name="flavor" id="flavor">
  <option value="0"></option>
  <optgroup label="Soft Flavors">
    <option value="softChoc">Chocolate</option>
    <option value="softStraw">Strawberry</option>
    <option value="softVan">Vanilla</option>
  </optgroup>
  <optgroup label="Hard Flavors">
    <option value="hardChoc">Chocolate</option>
    <option value="hardStraw">Strawberry</option>
    <option value="hardVan">Vanilla</option>
    <option value="hardMint">Mint Chocolate Chip</option>
    <option value="hardCC">Cookies &amp; Cream</option>
  </optgroup>
</select>
```

This will appear as follows in Google Chrome:

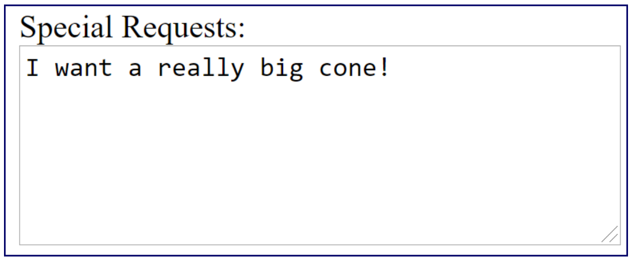


1.9. Textareas

Textareas are created with the `<textarea>` tag. Take a look at the following code:

```
<label for="requests">Special Requests:</label><br>
<textarea name="requests" id="requests" cols="40" rows="6"></textarea>
```

This would appear as follows after the user has entered data:



I want a really big cone!

The `cols` and `rows` attributes indicate the number of columns and rows (in characters) that the `textarea` should span.

An initial value can be entered into the `textarea` by adding text between the opening and closing `<textarea>` tags. For example:

```
<textarea name="requests" cols="40" rows="6">Hello, there!</textarea>
```

This is often done when a user submits the form with errors. The server-side code then returns the form already populated with the previously entered values, so the user doesn't have to re-enter them.

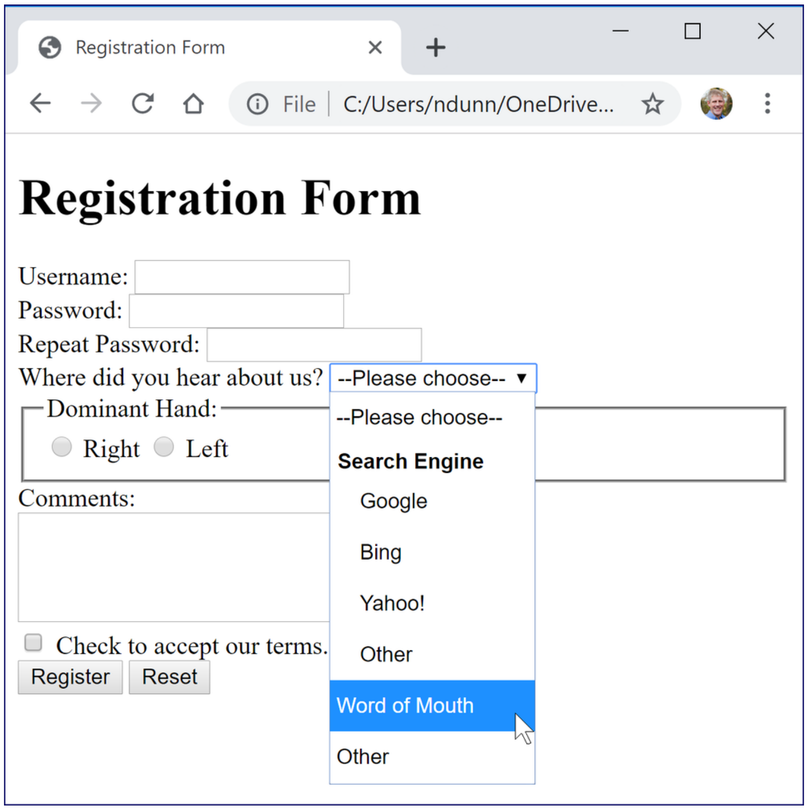
The `textarea` element's attributes are shown below:

1. `name` – Variable name.
2. `cols` – Width of textarea in average width of characters.
3. `rows` – Height of textarea in number of lines.
4. `minlength` – Minimum number of characters that must be entered.
5. `maxlength` – Maximum number of characters that can be entered.
6. `pattern` – A regular expression expressing a valid value for the field.
7. `placeholder` – A placeholder value that will disappear as soon as data is entered into the field.
8. `required` – When present, the user must fill in a value before submitting the form.

Exercise 3: Adding a Select Menu and a Textarea

15 to 25 minutes

In this exercise, you will add a select menu and textarea to the registration form. On completion, the form should look like this:



1. Open `Forms/Exercises/registration.html` for editing if you don't have it open already.
2. Add the following form elements:
   A. `select`: `name` and `id` should be "referral". Options should be broken up as follows (refer to the screenshot above):
      i. "--Please choose--" with value of "0"
      ii. An option group with the label of "Search Engine" and the following options:
         a. "Google" with value of "Google"

b. “Bing” with value of “Bing”

c. “Yahoo!” with value of “Yahoo”

d. “Other” with value of “OtherSearchEngine”

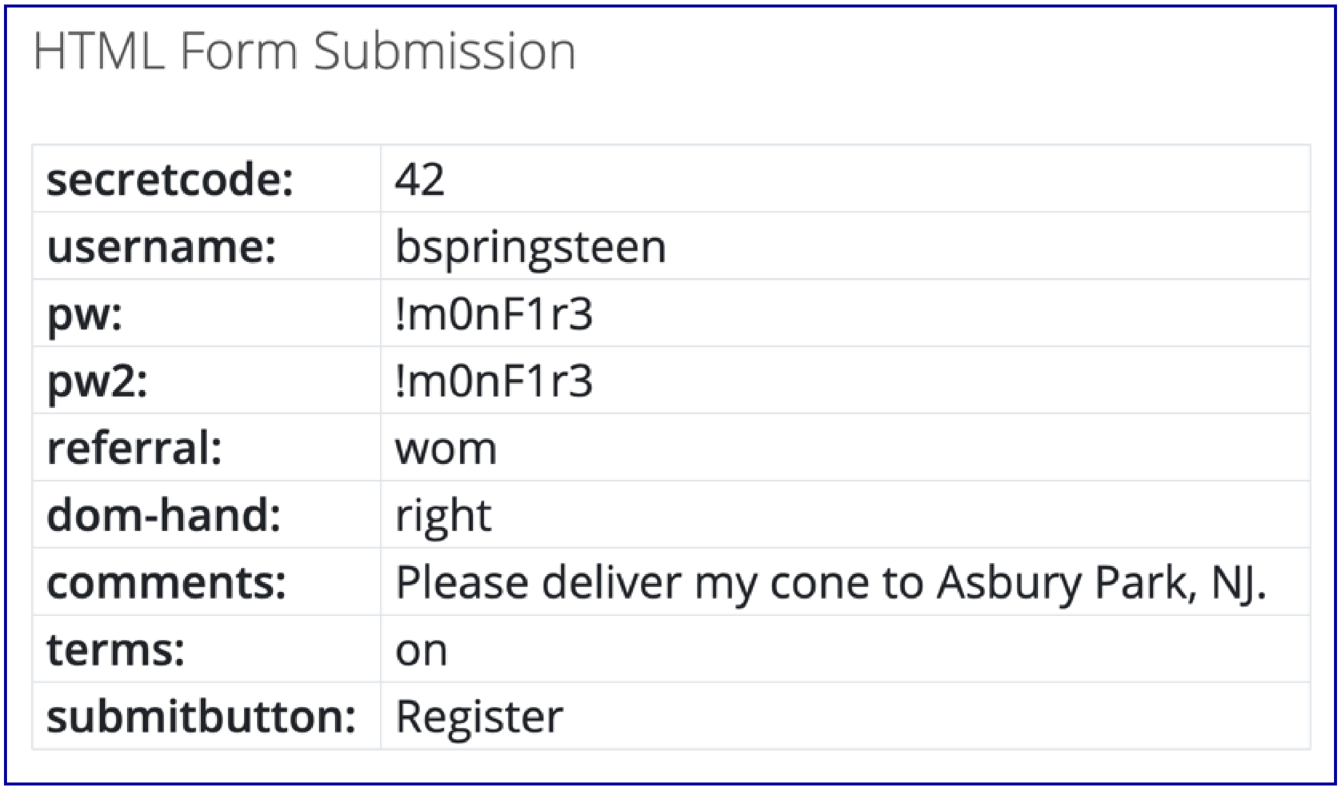
iii. “Word of Mouth” with value of “wom”

iv. “Other” with value of “other”

B. `textarea`: `name` and `id` should be “comments”. Be sure to include `cols` and `rows` attributes.

3. Put the “dominant hand” radio buttons in a `fieldset`.

When you are done, open the page in a browser and fill out and submit the form. You will need internet access to see the resulting page, which will look something like this:



HTML Form Submission

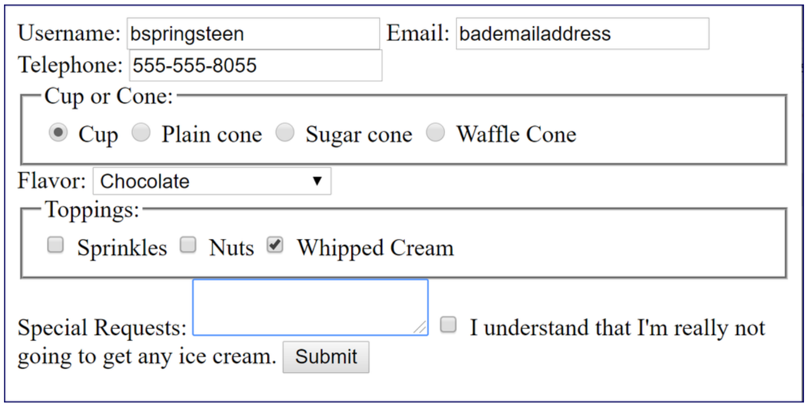
secretcode:	42
username:	bspringsteen
pw:	!m0nF1r3
pw2:	!m0nF1r3
referral:	wom
dom-hand:	right
comments:	Please deliver my cone to Asbury Park, NJ.
terms:	on
submitbutton:	Register

Solution: Forms/Solutions/registration-3.html

```
-------Lines 1 through 24 Omitted-------
<div>
  <label for="referral">Where did you hear about us?</label>
  <select name="referral" id="referral">
    <option value="0">--Please choose--</option>
    <optgroup label="Search Engine">
      <option value="Google">Google</option>
      <option value="Bing">Bing</option>
      <option value="Yahoo">Yahoo!</option>
      <option value="OtherSearchEngine">Other</option>
    </optgroup>
    <option value="wom">Word of Mouth</option>
    <option value="other">Other</option>
  </select>
</div>
<div>
<fieldset>
  <legend>Dominant Hand:</legend>
  <label>
    <input type="radio" name="dom-hand" id="hand-right" value="right">
    Right
  </label>
  <label>
    <input type="radio" name="dom-hand" id="hand-left" value="left">
    Left
  </label>
</fieldset>
</div>
<div>
  <label for="comments">Comments:</label><br>
  <textarea name="comments" id="comments"
    cols="40" rows="4"></textarea>
</div>
-------Lines 57 through 68 Omitted-------
```

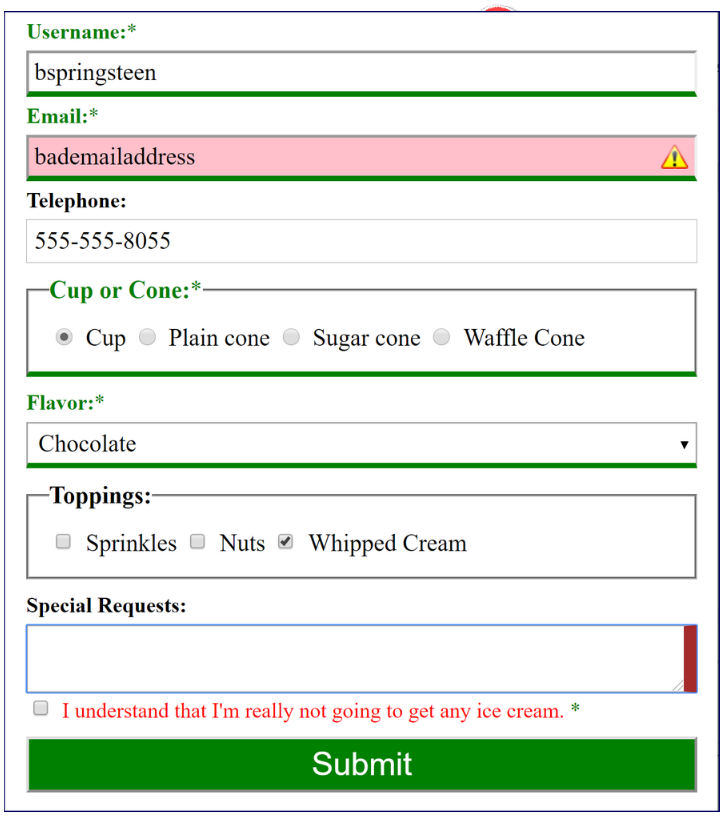
1.10. HTML Forms and CSS

As with all elements, forms can be laid out better and made to look much prettier with CSS. For example, here is a completed ice cream form without CSS:



Username: bspringsteen Email: bademailaddress
Telephone: 555-555-8055
Cup or Cone: [✓ Cup] Plain cone Sugar cone Waffle Cone
Flavor: Chocolate
Toppings: Sprinkles Nuts [✓ Whipped Cream]
Special Requests: I understand that I'm really not going to get any ice cream. Submit

And here is the same form with CSS added:



Username:*
bspringsteen
Email:*
bademailaddress
Telephone:
555-555-8055
Cup or Cone:* [✓ Cup] Plain cone Sugar cone Waffle Cone
Flavor:*
Chocolate
Toppings: Sprinkles Nuts [✓ Whipped Cream]
Special Requests:
I understand that I'm really not going to get any ice cream. *
Submit

Conclusion

In this lesson, you have learned to create HTML forms.

LESSON 2
JavaScript Form Validation

Topics Covered

☑ Accessing data entered by users in forms.

☑ Validating text fields, textareas, radio buttons, checkboxes, and select menus.

☑ Writing clean, reusable validation functions.

Introduction

In this lesson, you will learn to validate forms with HTML and JavaScript.

2.1. Server-side Form Validation

In most cases, when a user submits a registration form, a login form, a purchase form, or any other type of form, the data gets sent to a web server for processing. That data should be validated as soon as it hits the server, before doing anything else with it. This is true even if the data has already been validated on the client (the browser). That's because, it is impossible for the server to know that the client-side validation actually occurred. Nefarious hackers can easily get around JavaScript validation. In fact, it can be as simple as turning JavaScript off in the browser. To see how easy that is, simply google "Turn off JavaScript in Google Chrome" or in whichever browser you want to test. Server-side validation can be done with PHP, Java, Python, or any server-side programming language.

So, is it worth validating the code on the client if you have to do it again on the server anyway? Yes, it is. Here's why:

1. **Better user experience.** Client-side validation is practically immediate. In fact, it can be done as the user types. So, the user gets immediate feedback as to what needs to be corrected.
2. **Less work for your server.** If the data has already been validated on the client, then it is likely to pass server-side validation as well, meaning your server will only have to check it once.

2.2. HTML Form Validation

HTML form fields include validation attributes, such as `required`, `pattern`, `min` and `max`, and `minlength` and `maxlength`. In addition, setting the `type` to "email" or "url" forces the user to enter a valid email or URL. These are called constraints. While adding HTML constraints is very simple, it is difficult to control the user experience. To illustrate, let's look at an ice cream order form:

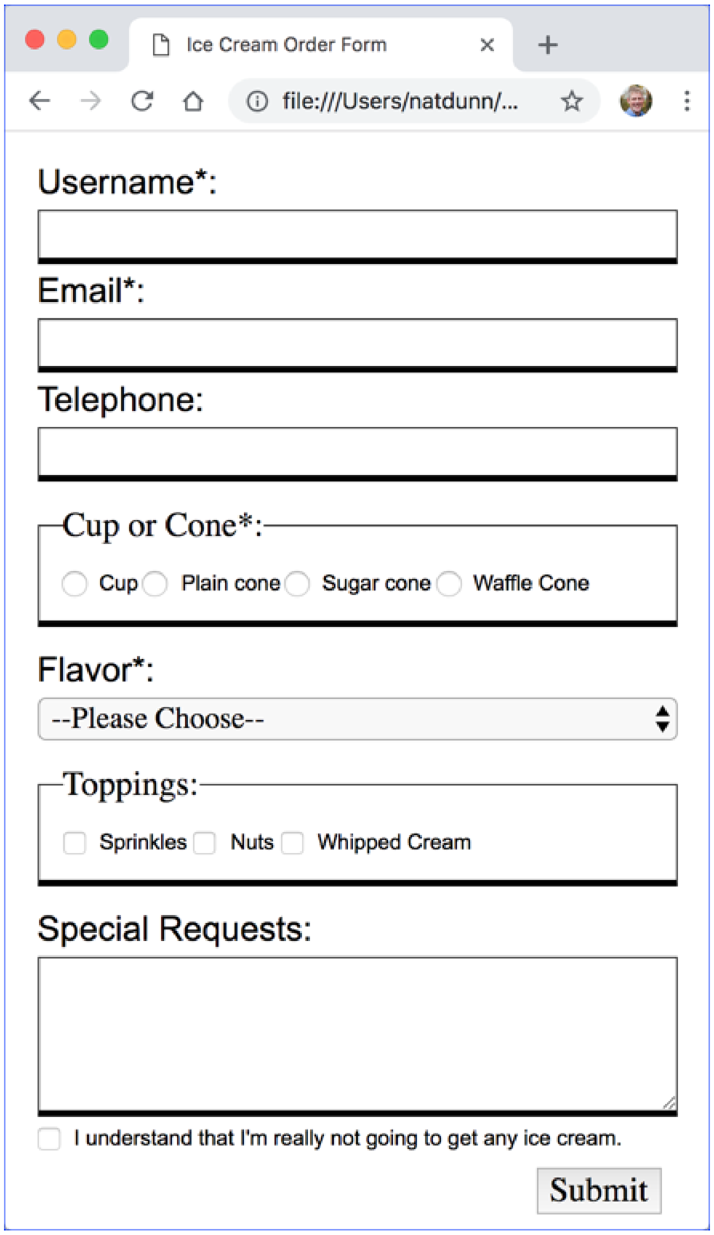
Demo 2.1: FormValidation/Demos/ice-cream.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="../normalize.css">
<link rel="stylesheet" href="../styles.css">
<title>Ice Cream Order Form</title>
</head>
<body>
<form id="ice-cream-form" method="post">
  <label for="username">Username*:</label>
  <input type="text" id="username" name="username"
    required minlength="8" maxlength="25">
  <label for="email">Email*:</label>
  <input type="email" id="email" name="email" required>
  <label for="phone">Telephone:</label>
  <input type="tel" id="phone" name="phone"
    pattern="[1-9]\d{2}-\d{3}-\d{4}">

  <fieldset>
    <legend>Cup or Cone*:</legend>
    <label>
      <input type="radio" name="container" value="cup" required>
      Cup
    </label>
    <label>
      <input type="radio" name="container" value="plaincone" required>
      Plain cone
    </label>
    <label>
      <input type="radio" name="container" value="sugarcone" required>
      Sugar cone
    </label>
    <label>
      <input type="radio" name="container" value="wafflecone" required>
      Waffle Cone
    </label>
  </fieldset>

  <label for="flavor">Flavor*:</label>
  <select name="flavor" id="flavor" required>
    <option value="0" selected="selected">--Please Choose--</option>
    <optgroup label="Soft Flavors">
```

```
      <option value="softChoc">Chocolate</option>
      <option value="softStraw">Strawberry</option>
      <option value="softVan">Vanilla</option>
    </optgroup>
    <optgroup label="Hard Flavors">
      <option value="hardChoc">Chocolate</option>
      <option value="hardStraw">Strawberry</option>
      <option value="hardVan">Vanilla</option>
      <option value="hardMint">Mint Chocolate Chip</option>
      <option value="hardCC">Cookies &amp; Cream</option>
    </optgroup>
  </select>

  <fieldset id="toppings">
    <legend>Toppings:</legend>
    <input type="checkbox" name="sprinkles" id="sprinkles">
    <label for="sprinkles">Sprinkles</label>
    <input type="checkbox" name="nuts" id="nuts">
    <label for="nuts">Nuts</label>
    <input type="checkbox" name="whip" id="whip">
    <label for="whip">Whipped Cream</label>
  </fieldset>

  <label for="requests">Special Requests:</label>
  <textarea name="requests" id="requests"
    minlength="10" maxlength="200"></textarea><br>

  <input type="checkbox" name="terms" id="terms" required>
  <label for="terms">
    I understand that I'm really not going to get any ice cream.
  </label>

  <button>Submit</button>
</form>
</body>
</html>
```

Code Explanation

We have added some basic CSS to style the form:


Ice Cream Order Form

file:///Users/natdunn/...

Username*:

Email*:

Telephone:

Cup or Cone*:

○ Cup ○ Plain cone ○ Sugar cone ○ Waffle Cone

Flavor*:

--Please Choose--

Toppings:

☐ Sprinkles ☐ Nuts ☐ Whipped Cream

Special Requests:

☐ I understand that I'm really not going to get any ice cream.

Submit

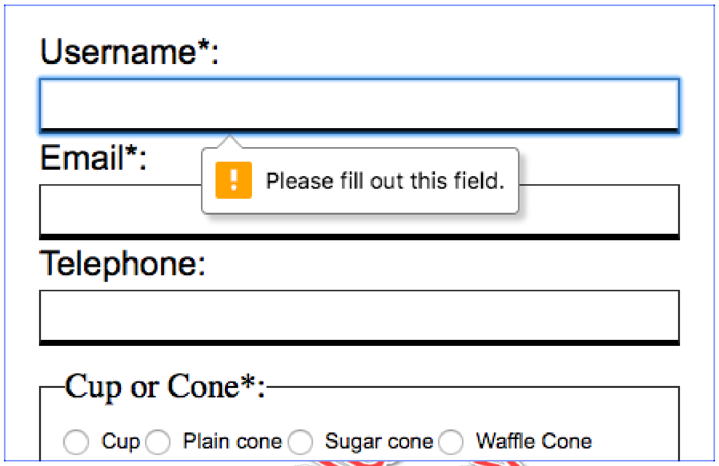
This form has no JavaScript validation, but the form fields do include HTML constraints:

1. The `username` must be between 8 and 25 characters and is required.
2. The `email` must be a valid email address and is required.
3. The `phone` must match a regular expression[5] if it is entered. It is not required.

5. Regular expressions are used by many programming languages for pattern matching.

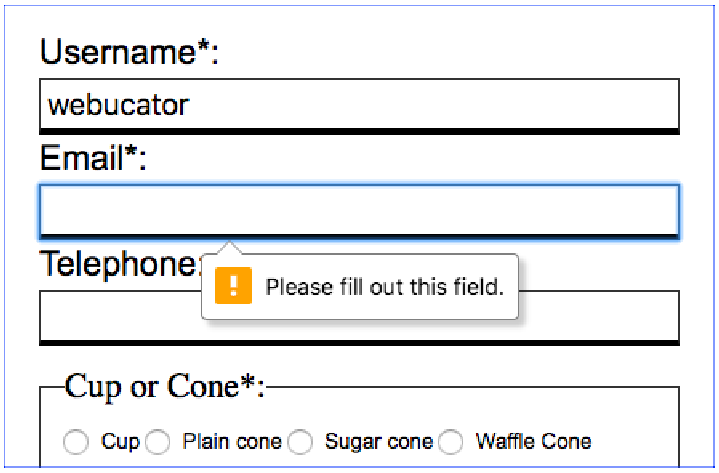
4. A `container` selection is required.
5. A `flavor` selection is required.
6. A `requests` entry must be between 10 and 200 characters if it is entered. It is not required.
7. The `terms` checkbox must be checked.

While this is functional, it's not very user friendly. Watch what happens when we submit the form without entering any data:



Notice that, while both the `username` and the `email` fields are invalid, we only get an error for the `username` field.

Now let's fill in a `username` and resubmit:



This time we get an error on the `email` field, but not on any of the other fields.

So, the user only finds out one error at a time and then must attempt to submit after each correction.

JavaScript to the rescue! With JavaScript, we can enhance and customize the validation of forms.

2.3. Accessing Form Data

All forms on a web page are stored in the `document.forms[]` array. The first form on a page is `document.forms[0]`, the second form is `document.forms[1]`, and so on. However, it is usually easier to reference forms via their `id` attribute and refer to them that way. For example, a form with id `login-form` can be referenced as `document.getElementById('login-form')`. The major advantage of referencing forms by `id` is that the forms can be repositioned on the page without affecting the JavaScript.

Elements within a form are properties of that form and can be referenced as follows, where `elementName` is the value of the element's `name` attribute:

```
document.getElementById('login-form').elementName
```

For example, given the following form:

```
<form id="login-form">
   <label for="uname">Username:</label>
   <input type="text" name="username" id="uname">
   <input type="submit" value="Log in">
</form>
```

You can access the `username` field like this:

```
const username = document.getElementById('login-form').username;
```

As with all elements, you can also access a field by its `id`:

```
const username = document.getElementById('uname');
```

Names and IDs

It is common to use the same value for both the `name` and `id` of form fields. In the example above, we use different values simply to distinguish between the examples of accessing the field by `name` and `id`.

All text-like fields (e.g., `text`, `password`, `url`, `email`, `tel`, etc.) have a `value` property that holds the text value of the field. Take a look at the following example:

Demo 2.2: FormValidation/Demos/text-like-fields.html

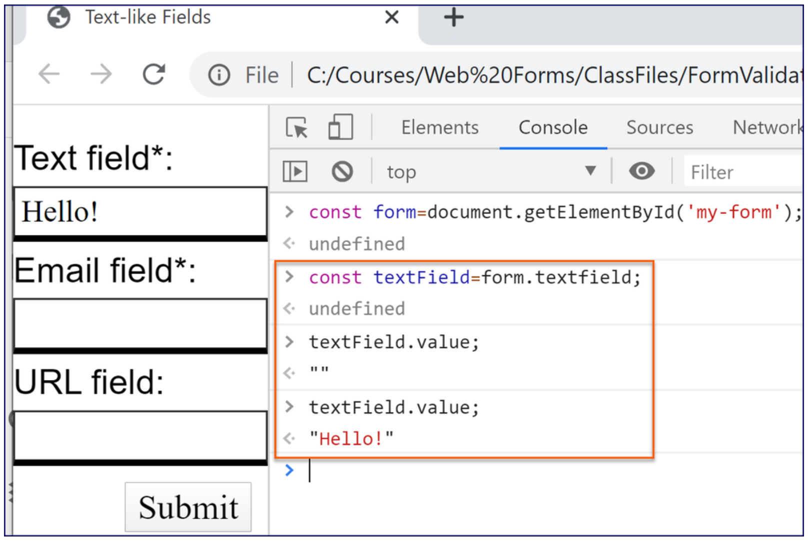
```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="../normalize.css">
<link rel="stylesheet" href="../styles.css">
<title>Text-like Fields</title>
</head>
<body>
<form id="my-form" method="post" novalidate>
  <label for="textfield">Text field*:</label>
  <input type="text" id="textfield" name="textfield"
    required minlength="5" maxlength="10">
  <label for="emailfield">Email field*:</label>
  <input type="email" id="emailfield" name="emailfield" required>
  <label for="urlfield">URL field:</label>
  <input type="url" id="urlfield" name="urlfield" pattern="https:.*">

  <button>Submit</button>
</form>
</body>
</html>
```

Code Explanation

1. Open `FormValidation/Demos/text-like-fields.html` in the browser, open the console, and type:

```
const form = document.getElementById('my-form');
const textField = form.textfield;
textField.value;
```

2. You will see the above code outputs an empty string. That is because the field has no text in it.
3. Now type "Hello!" in the text field and type `textField.value;` in the console again. This time, `textField.value` returns "Hello!":



```
> const form=document.getElementById('my-form');
<· undefined
> const textField=form.textfield;
<· undefined
> textField.value;
<· ""
> textField.value;
<· "Hello!"
>
```

4. All values of text-like fields can be accessed in this way. Try it with `emailfield` and `urlfield`.

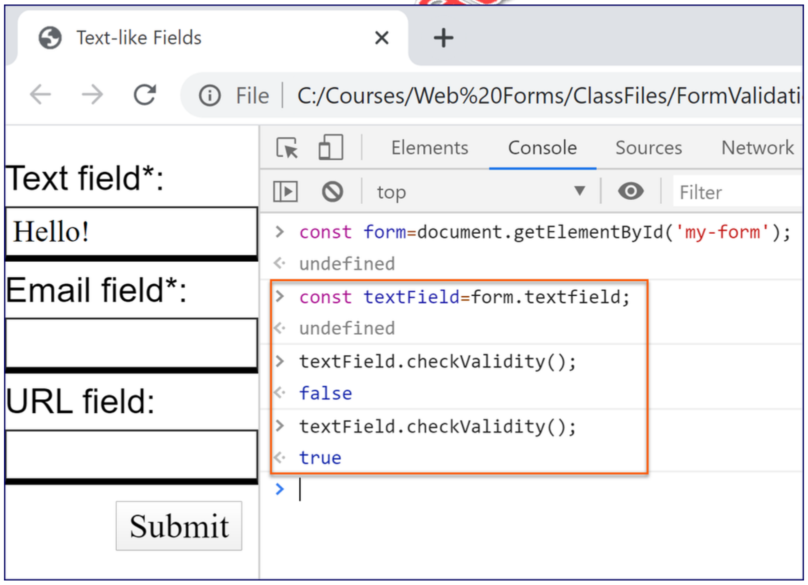
2.4. Form Validation with JavaScript

The first step to using JavaScript to validate your forms is to turn off the default HTML validation. You do this by adding the `novalidate` attribute to the `<form>` tag:

```
<form id="my-form" method="post" novalidate>
```

This prevents the default HTML validation from happening when the form is submitted, so you can use JavaScript to create custom validation.

To check if a field's value is valid, use the `checkValidity()` method of the field. In the following screenshot, we first ran `textField.checkValidity();` when the field was empty and then ran it again after typing "Hello!" into the text field:



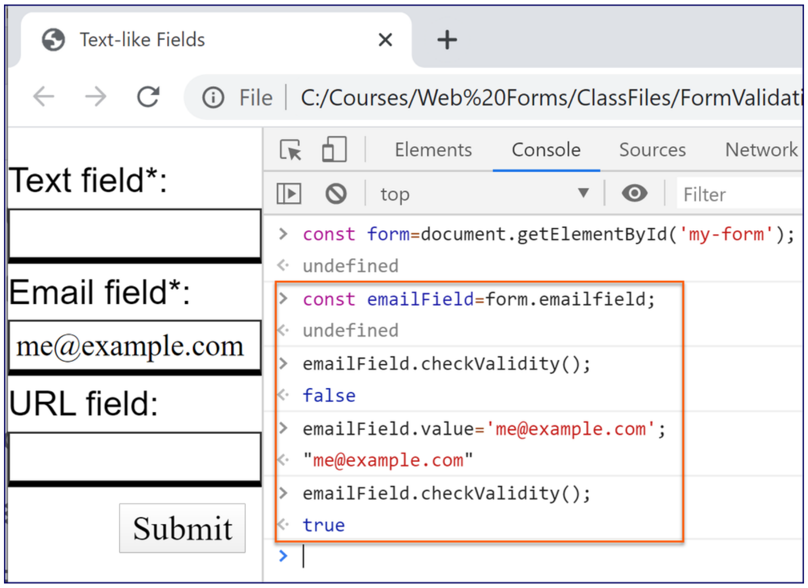
```
> const form=document.getElementById('my-form');
<· undefined
> const textField=form.textfield;
<· undefined
> textField.checkValidity();
<· false
> textField.checkValidity();
<· true
>
```

Exercise 4: Checking the Validity of the Email and URL Fields

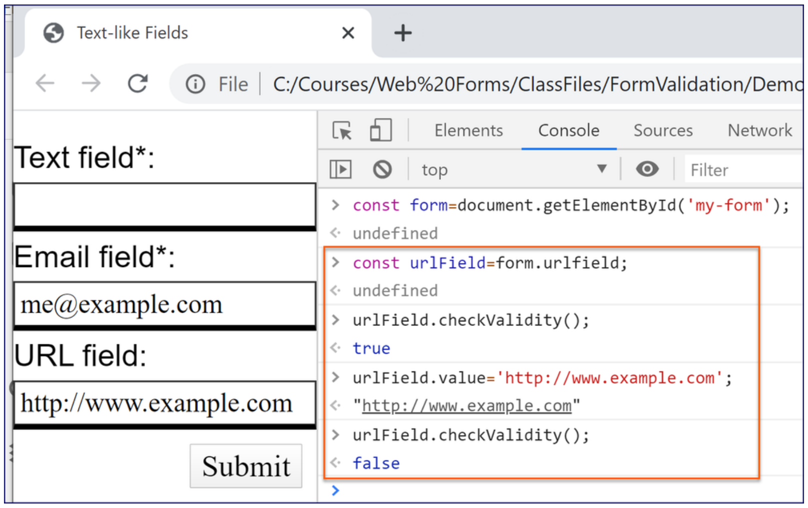
5 to 10 minutes

1. From your `Demos` folder, open `text-like-fields.html` in your browser, if it's not open already.
2. In the console, check the validity of the `emailfield` and `urlfield` fields. If the field is **valid** to start, modify the field value to make it **invalid** and check it again. If it is **invalid** to start, modify the field value to make it **valid** and check it again.
3. Note that `emailfield` is required and must be a well-formed email, and `urlfield` is not required, but if entered, it must be a valid URL beginning with "https:".

Solution


Text-like Fields
File | C:/Courses/Web%20Forms/ClassFiles/FormValidati
Elements Console Sources Network
top
Filter
Text field*:
Email field*:
me@example.com
URL field:
Submit

```
> const form=document.getElementById('my-form');
<· undefined
> const emailField=form.emailfield;
<· undefined
> emailField.checkValidity();
<· false
> emailField.value='me@example.com';
<· "me@example.com"
> emailField.checkValidity();
<· true
```



```
> const form=document.getElementById('my-form');
<· undefined
> const urlField=form.urlfield;
<· undefined
> urlField.checkValidity();
<· true
> urlField.value='http://www.example.com';
<· "http://www.example.com"
> urlField.checkValidity();
<· false
```

The `url` field isn't valid in the last example, because it starts with "http:" and the `pattern` requires that it starts with "https:".

2.5. Checking Validity on Input and Submit Events

For the best user experience, in addition to checking all fields when the user submits the form, we should check the validity of each field when the user changes the value of that field. To do this, we will listen for `submit` events on the form, and `input` events on the text-like fields. We will add those listeners as soon as the page loads. We can use the following function to check the validity of a field and turn the background color of the control to pink if the entry is invalid:

```
function checkField(field) {
  if (!field.checkValidity()) {
    field.style.backgroundColor = 'pink';
  } else {
    field.style.backgroundColor = '';
  }
}
```

We will call the `checkField()` function when the user inputs a value in the field we want to check and when the user submits the form. To do that, we add our event listeners:

```
window.addEventListener('load', function(e) {
  const form = document.getElementById('my-form');
  const textField = form.textfield;

  textField.addEventListener('input', function(e) {
    checkField(textField);
  });

  form.addEventListener("submit", function(e) {
    // Check errors
    checkField(textField);

    // If form is invalid, prevent submission
    if (!form.checkValidity()) {
      e.preventDefault();
      alert('Please fix form errors.');
    }
  });
});
```

A few things to note:

1. The event we are listening for is an `input` event. This fires whenever the value of an `input`, `select`, or `textarea` changes; however, at the time of this writing, it is only reliable for text-like `input` and `textarea` fields.
   A. For checkboxes and radio buttons, it is better to listen for `click` events.
   B. For `select` fields, it is better to listen for `change` events.
2. For now, we're just changing the background color of invalid fields to pink. Later, we'll add useful messages.
3. `form` elements also take the `checkValidity()` method, which returns `false` if any of the form's fields are invalid. Otherwise, it returns `true`.
4. The `preventDefault()` method of an event is used to prevent what would normally happen when that event occurs. In this case, the event is `submit`, which would normally cause the form to be submitted. To prevent that from happening when the form is invalid, we call `e.preventDefault()`.
5. We include an `alert()` call only for illustration purposes. In practice, you probably don't need the alert. You can just show the errors in the form.

Input Event

The `input` event is meant to be fired when the user changes the value of `input`, `select`, and `textarea` elements, but at the time of this writing, it is not yet fully reliable. Check `https://caniuse.com/#search=input` for information on current browser support.

Below, we have the complete file:

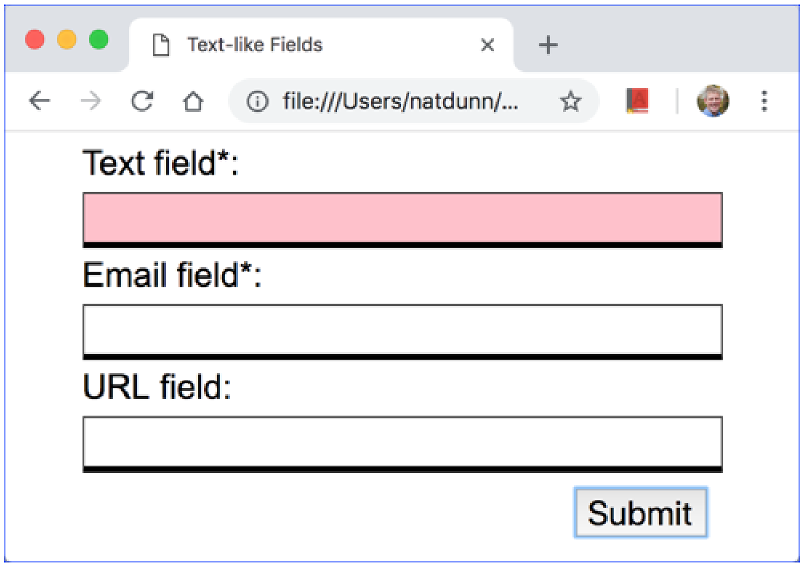
Demo 2.3: FormValidation/Demos/text-like-fields-validate.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="../normalize.css">
<link rel="stylesheet" href="../styles.css">
<script>
    function checkField(field) {
    if (!field.checkValidity()) {
      field.style.backgroundColor = 'pink';
    } else {
      field.style.backgroundColor = '';
    }
  }

  window.addEventListener('load', function(e) {
    const form  = document.getElementById('my-form');
    const textField = form.textfield;

    textField.addEventListener('input', function(e) {
      checkField(textField);
    });

    form.addEventListener('submit', function(e) {
      // Check errors
      checkField(textField);

      // If form is invalid, prevent submission
      if (!form.checkValidity()) {
        e.preventDefault();
        alert('Please fix form errors.');
      }
    });
  });
</script>
<title>Text-like Fields</title>
</head>
<body>
<form id="my-form" method="post" novalidate>
  <label for="textfield">Text field*:</label>
  <input type="text" id="textfield" name="textfield"
    required minlength="5" maxlength="10">
  <label for="emailfield">Email field*:</label>
```

```
    <input type="email" id="emailfield" name="emailfield" required>
    <label for="urlfield">URL field:</label>
    <input type="url" id="urlfield" name="urlfield" pattern="https:.*">

    <button>Submit</button>
  </form>
  </body>
  </html>
```

Code Explanation

1. Open `FormValidation/Demos/text-like-fields-validate.html` in your browser and submit the form without filling in any of the fields. The text field should turn pink:



2. Enter a value of at least 5 characters in the text field. You will see that the pink background goes away.

Now let's add validation for the `email` and `url` fields:

Demo 2.4: FormValidation/Demos/text-like-fields-validate2.html

```
-------Lines 1 through 16 Omitted-------
  window.addEventListener('load', function(e) {
    const form  = document.getElementById('my-form');
    const textField = form.textfield;
    const emailField = form.emailfield;
    const urlField = form.urlfield;

    textField.addEventListener('input', function(e) {
      checkField(textField);
    });

    emailField.addEventListener('input', function(e) {
      checkField(emailField);
    });

    urlField.addEventListener('input', function(e) {
      checkField(urlField);
    });

    form.addEventListener('submit', function(e) {
      // Check errors
      checkField(textField);
      checkField(emailField);
      checkField(urlField);

      // If form is invalid, prevent submission
      if (!form.checkValidity()) {
        e.preventDefault();
        alert('Please fix form errors.');
      }
    });
  });
-------Lines 48 through 64 Omitted-------
```

Code Explanation

1. Notice that we reuse the `checkField()` function without modification.
2. Open `FormValidation/Demos/text-like-fields-validate2.html`in your browser and submit the form without filling in any of the fields. You will notice that only the Text and Email fields appear pink. That's because the URL field is not required.
3. Enter an invalid URL in the URL field (e.g., "hello") and resubmit. This time all three fields will appear pink.

2.6. Adding Error Messages

There are many ways to add error messages to the form. One common practice is to have the error messages within the HTML, but to hide them until it is time to report an error. You could do this by setting the `display` property to “none” by default and then changing it to “block” to report the error. One downside to this method is that it makes the HTML messy.

Instead of adding error messages to our HTML that may never be shown to the user, we will use JavaScript to dynamically add elements containing error messages.

❖ 2.6.1. The dataset Property

HTML elements can take a `dataset` property to associate non-standard data with an element. We will use this to assign error messages to our form fields. The following code illustrates this:

```
const form = document.getElementById('my-form');
const textField = form.textfield;
textField.dataset.errorMsg = 'You must enter a value.';
```

Sometimes it is useful to use other properties of the field in the error message. For example, our `textField` field is created with the following HTML:

```
<input type="text" id="textfield" name="textfield"
  required minlength="5" maxlength="10">
```

Below we use the `minLength` and `maxLength` properties of `textField` to create the error message. Note that JavaScript uses camelCase for the property names (e.g., `minLength` and `maxLength`):

```
textField.dataset.errorMsg = 'Your entry must be between ' +
  textField.minLength + ' and ' +
  textField.maxLength + ' characters.';
```

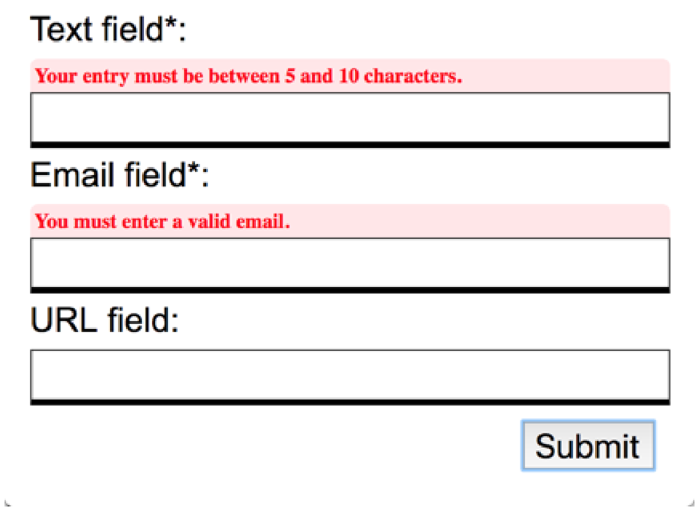
Now we’ll look at how we can report these custom errors to the user in a friendly way:

Demo 2.5: FormValidation/Demos/text-like-fields-add-error1.html

```
  -------Lines 1 through 7 Omitted-------
    <script>
      function addError(field) {
      const error = document.createElement('div');
      error.innerHTML = field.dataset.errorMsg;
      error.className = 'error';
      field.parentNode.insertBefore(error, field);
    }

    function removeError(field) {
      // to do
    }

    function checkField(field) {
      if (!field.checkValidity()) {
        addError(field);
      } else {
        removeError(field);
      }
    }

    window.addEventListener('load', function(e) {
      const form  = document.getElementById('my-form');
      const textField = form.textfield;
      textField.dataset.errorMsg = 'Your entry must be between ' +
      textField.minLength + ' and ' + textField.maxLength +
      ' characters.';

      const emailField = form.emailfield;
      emailField.dataset.errorMsg = 'You must enter a valid email.';

      const urlField = form.urlfield;
      urlField.dataset.errorMsg = 'The URL must begin with "https".';

      textField.addEventListener('input', function(e) {
        checkField(textField);
      });

      emailField.addEventListener('input', function(e) {
        checkField(emailField);
      });

      urlField.addEventListener('input', function(e) {
        checkField(urlField);
      });

      form.addEventListener('submit', function(e) {
        // Check errors
        checkField(textField);
        checkField(emailField);
        checkField(urlField);

        // If form is invalid, prevent submission
        if (!form.checkValidity()) {
          e.preventDefault();
          alert('Please fix form errors.');
        }
      });
    });
  </script>
  -------Lines 67 through 82 Omitted-------
```

Code Explanation

1. Notice that we have added `errorMsg` properties to the `dataset` properties of our field elements on the `load` event.
2. Notice that we have changed `checkField()` to call `addError(field)` and `removeError(field)` instead of setting and unsetting a pink background.
3. Open `FormValidation/Demos/text-like-fields-add-error1.html` in your browser and submit the form without filling in any of the fields. After seeing an alert warning you that there are errors, you should see those errors appear right above the two empty required input fields:


Text field*:
Your entry must be between 5 and 10 characters.
Email field*:
You must enter a valid email.
URL field:
Submit

4. The `addError()` function takes one argument: the `field` to which to add an error message. That field should be sent to the function with an `errorMsg` property in its `dataset`. The function does the following:

   A. Creates a new `div` element and assigns it to `error`:

   ```
   const error = document.createElement('div');
   ```

   B. Sets the `innerHTML` of the `error` to the field's `errorMsg`:

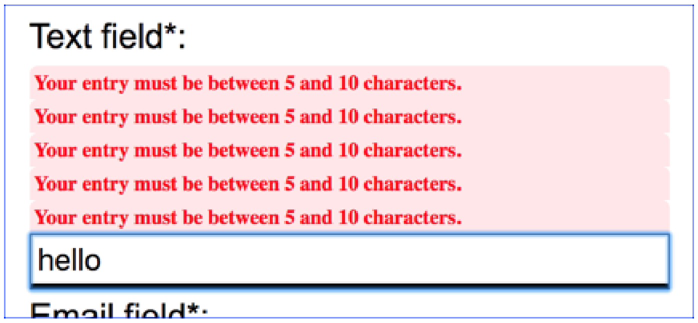
   ```
   error.innerHTML = field.dataset.errorMsg;
   ```

   C. Sets the `className` of `error` to `'error'`:

   ```
   error.className = 'error';
   ```

   D. Inserts the new `error` element before the passed-in `field`, so that the error shows up right above the field itself.

5. Now, type "hello" into the `textfield` input:


Text field*:

Your entry must be between 5 and 10 characters.
Your entry must be between 5 and 10 characters.
Your entry must be between 5 and 10 characters.
Your entry must be between 5 and 10 characters.
Your entry must be between 5 and 10 characters.

hello

Email field*:

6. It validates the field with each user input. And each time it finds the field invalid, it adds another `error div`. To prevent that from happening, we should check to see if the `error div` already exists and, if it does, exit the function without adding another `error div`. The following example does that:

Demo 2.6: FormValidation/Demos/text-like-fields-add-error2.html

```
    -------Lines 1 through 8 Omitted-------
      function addError(field) {
        if (field.previousElementSibling &&
          field.previousElementSibling.className === 'error') {
          // error message already showing
          return;
        }
        const error = document.createElement('div');
        error.innerHTML = field.dataset.errorMsg;
        error.className = 'error';
        field.parentNode.insertBefore(error, field);
    }
    -------Lines 20 through 85 Omitted-------
```

Code Explanation

Now the `addError()` function first checks to see if the passed-in field has a `previousElementSibling` and, if it does, if that element's `className` is "error". If it is, that means the `error div` is already present, so we return. We don't actually return anything. We are simply exiting the function and effectively passing the ball back to the code that called the function.

Open `FormValidation/Demos/text-like-fields-add-error2.html` in your browser to test it out.

Now we need to write our `removeError()` function so that we can remove the error when the user corrects the problem:

Demo 2.7: FormValidation/Demos/text-like-fields-remove-error.html

```
-------Lines 1 through 8 Omitted-------
    function addError(field) {
    if (field.previousElementSibling &&
      field.previousElementSibling.className === 'error') {
      // error message already showing
      return;
    }
    const error = document.createElement('div');
    error.innerHTML = field.dataset.errorMsg;
    error.className = 'error';
    field.parentNode.insertBefore(error, field);
  }

  function removeError(field) {
    if (field.previousElementSibling &&
      field.previousElementSibling.className === 'error') {
      field.previousElementSibling.remove();
    }
  }
-------Lines 27 through 88 Omitted-------
```

Code Explanation

1. Like with the `addError()` function, the `removeError()` function first checks to see if the passed-in field has a `previousElementSibling` and, if it does, if that element's `className` is "error". If it is, that means the `error div` is present, so we remove it using the `remove()` method.
2. Now, type "Hello" into the `textfield` input. The error should disappear as soon as the entry is valid.
3. Add and remove values from the other fields as well to see if they work as expected.

2.7. Validating Textareas

Validating `textarea` elements is similar to validating text-like `input` fields. In the following code sample we have added a `textarea`, which is not required, but if included must be between 10 and 200 characters:

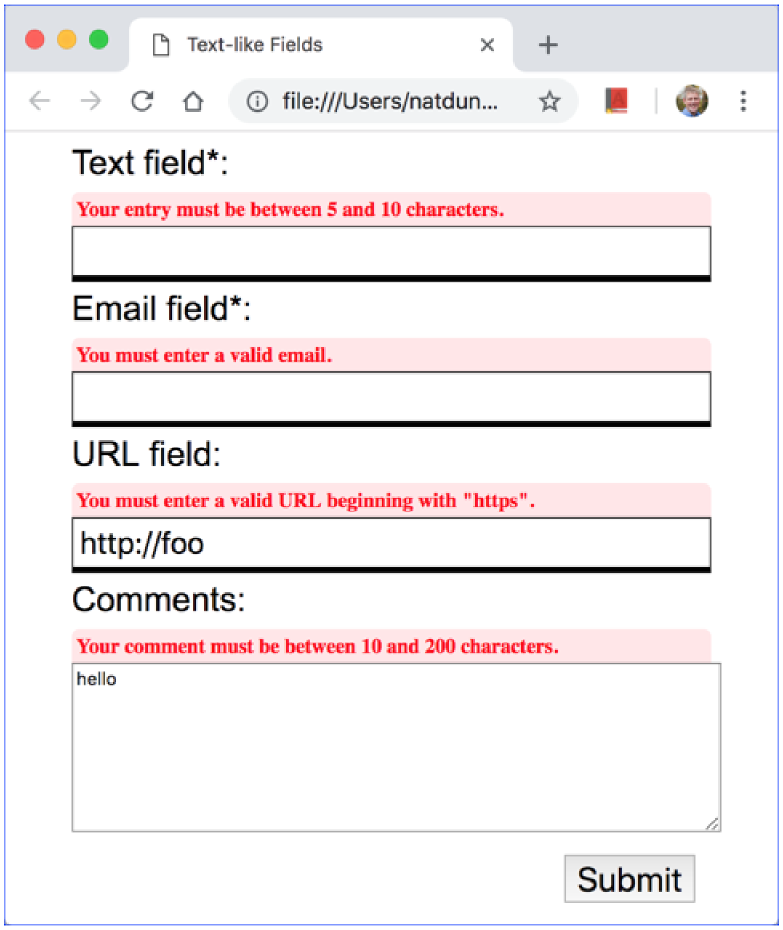
Demo 2.8: FormValidation/Demos/text-like-fields-complete.html

```
     -------Lines 1 through 46 Omitted-------
47.      const urlField = form.urlfield;
48.      urlField.dataset.errorMsg = 'The URL must begin with "https".';
49.
50.      const comments = form.comments;
51.      comments.dataset.errorMsg = 'Your comment must be between ' +
52.        comments.minLength + ' and ' + comments.maxLength +
53.        ' characters.';
54.
55.      textField.addEventListener('input', function(e) {
56.        checkField(textField);
57.      });
58.
59.      emailField.addEventListener('input', function(e) {
60.        checkField(emailField);
61.      });
62.
63.      urlField.addEventListener('input', function(e) {
64.        checkField(urlField);
65.      });
66.
67.      comments.addEventListener('input', function(e) {
68.        checkField(comments);
69.      });
70.
71.      form.addEventListener('submit', function(e) {
72.        // Check errors
73.        checkField(textField);
74.        checkField(emailField);
75.        checkField(urlField);
76.        checkField(comments);
77.
78.        // If form is invalid, prevent submission
79.        if (!form.checkValidity()) {
80.          e.preventDefault();
81.          alert('Please fix form errors.');
82.        }
83.      });
84.    });
     -------Lines 85 through 97 Omitted-------
98.    <label for="comments">Comments:</label>
99.    <textarea name="comments" id="comments"
100.     minlength="10" maxlength="200"></textarea><br>
```

```
101.
     -------Lines 102 through 105 Omitted-------
```

Code Explanation

The image belows shows how the form appears when all fields are invalid:



Text field*:

Your entry must be between 5 and 10 characters.

Email field*:

You must enter a valid email.

URL field:

You must enter a valid URL beginning with "https".

http://foo

Comments:

Your comment must be between 10 and 200 characters.

hello

Submit

2.8. Validating Checkboxes

Like the text-like fields we've been looking at, checkboxes are created with the `input` tag. We can reuse the same `checkField()` function we have created; however, rather than listening for `input` events, we will listen for `click` events.

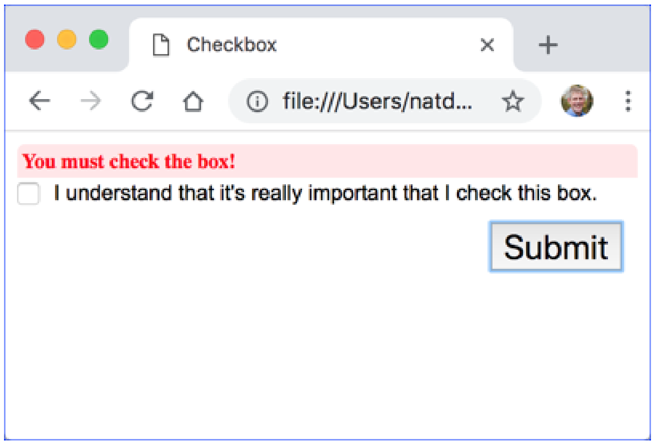
Demo 2.9: FormValidation/Demos/checkbox.html

```
-------Lines 1 through 35 Omitted-------
  window.addEventListener('load', function(e) {
    const form  = document.getElementById('my-form');
    const cb = form.terms;
    cb.dataset.errorMsg = 'You must check the box!';

    cb.addEventListener('click', function(e) {
      checkField(cb);
    });

    form.addEventListener("submit", function(e) {
      // Check errors
      checkField(cb);

      // If form is invalid, prevent submission
      if (!form.checkValidity()) {
        e.preventDefault();
        alert('Please fix form errors.');
      }
    });
  });
-------Lines 56 through 59 Omitted-------
<form id="my-form" method="post" novalidate>
  <input type="checkbox" name="terms" id="terms" required>
  <label for="terms">
    I understand that it's really important
    that I check this box.
  </label>
-------Lines 66 through 70 Omitted-------
```

Code Explanation

1. This code should all be clear as it's similar to the text-field validation in the earlier examples.

2. Open `FormValidation/Demos/checkbox.html` and submit the form without checking the checkbox. You should get an alert saying "Please fix form errors." and see an error above the checkbox:



You must check the box!

I understand that it's really important that I check this box.

3. As you check and uncheck the box, the error should disappear and reappear.

2.9. Validating Radio Buttons

Radio buttons are similar to checkboxes; however, since our `checkField()` function only expects a single element, we have to pass just one radio button `input` element to the function. Luckily, if any one of a group of radio buttons is invalid, they are all invalid, so we can just send the first radio button, which we can get like this:

```
const answer = form.answer[0];
answer.dataset.errorMsg = 'Please answer the question.';
```

As with checkboxes, we will listen for `click` events. We loop through `form.answer`, which is an array of radio buttons, adding the click event to each one:

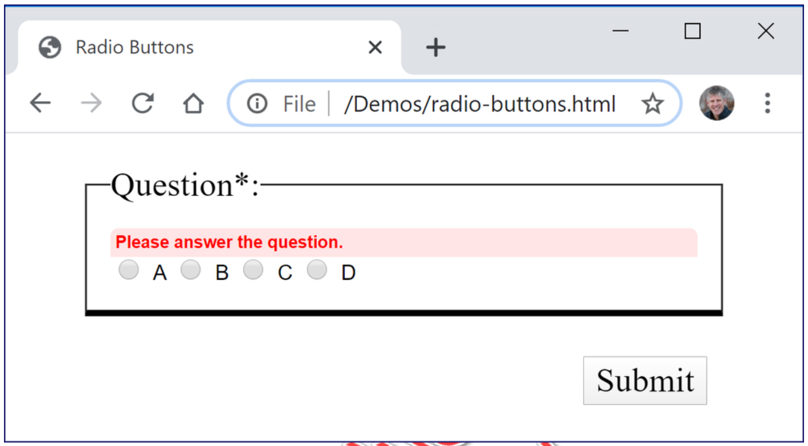
Demo 2.10: FormValidation/Demos/radio-buttons.html

```
-------Lines 1 through 35 Omitted-------
  window.addEventListener('load', function(e) {
    const form  = document.getElementById('my-form');
    const answer = form.answer[0];
    answer.dataset.errorMsg = 'Please answer the question.';

    // Add click event handler to each radio button
    for (let button of form.answer) {
      button.addEventListener('click', function(e) {
        checkField(answer);
      });
    }

    form.addEventListener("submit", function(e) {
      // Check errors
      checkField(answer);

      // If form is invalid, prevent submission
      if (!form.checkValidity()) {
        e.preventDefault();
        alert('Please fix form errors.');
      }
    });
  });
</script>
<title>Radio Buttons</title>
</head>
<body>
<form id="my-form" method="post" novalidate>
  <fieldset>
    <legend>Question*:</legend>
    <label>
      <input type="radio" name="answer" value="1" required> A
    </label>
    <label>
      <input type="radio" name="answer" value="2" required> B
    </label>
    <label>
      <input type="radio" name="answer" value="3" required> C
    </label>
    <label>
      <input type="radio" name="answer" value="4" required> D
    </label>
  </fieldset>
```

```
-------Lines 79 through 83 Omitted-------
```

Code Explanation

1. This code should all be clear as it's similar to the text-field validation in the earlier examples.
2. Open `FormValidation/Demos/radio-buttons.html` and submit the form without selecting an answer. You should get an alert to fix form errors and see an error message:



Please answer the question.

3. As soon as you make a selection, the error should disappear.

❖ 2.9.1. Which Radio Button was Selected?

The following function is useful for checking which radio button in a group of radio buttons is selected. This might be useful in an upcoming challenge exercise!

```
function getSelectedRadio(radioArray) {
  for (btn of radioArray) {
    if (btn.checked) {
      return btn;
    }
  }
  return null;
}
```

The function simply loops through the buttons until it finds one that is checked, at which point it returns that button. If it doesn't find any checked buttons, it returns null.

2.10. Validating Select Menus

When select menus allow for only one option to be selected (i.e., when the `multiple` attribute is not used), then there is always a default selection. If it is not set specfically using the `selected` attribute, then the first option (at index 0) will be selected by default. As such, the `required` attribute, while valid, doesn't actually change anything. It is a common practice to make the first option of `select` menu an invalid option that might be blank or read "--Please choose--"; however, there is no way to enforce this with HTML alone. Using JavaScript, we can report an error if the first option is left selected. In addition, because leaving the first option selected doesn't make the `select` menu invalid, we need to explicitly make the field invalid. This is done with the element's `setCustomValidity()` method, which takes a string with an error message:

```
function checkSelect(field) {
  if ( field.selectedIndex === 0 ) {
    field.setCustomValidity('Invalid');
    addError(field);
  } else {
    field.setCustomValidity('');
    removeError(field);
  }
}
```

By setting the custom validity to a string with at least one character, we make the field invalid, which in turn makes the whole form invalid. Then in the else block, we set the custom validity to an empty string, which indicates that the element has no custom validity error.

The following example shows how we incorporate this `checkSelect()` function in our form validation:

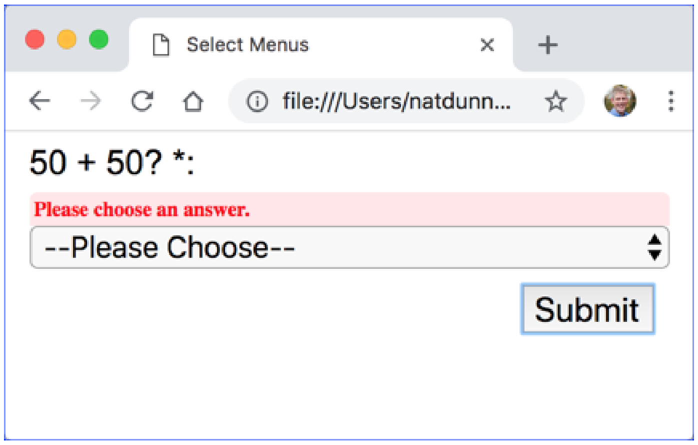
Demo 2.11: FormValidation/Demos/select-menus.html

```
-------Lines 1 through 27 Omitted-------
 function checkSelect(field) {
    if ( field.selectedIndex === 0 ) {
      field.setCustomValidity('Invalid');
      addError(field);
    } else {
      field.setCustomValidity('');
      removeError(field);
    }
  }

  window.addEventListener('load', function(e) {
    const form  = document.getElementById('my-form');
    const problem = form.problem;
    problem.dataset.errorMsg = 'Please choose an answer.';

    problem.addEventListener('change', function(e) {
      checkSelect(problem);
    });

    form.addEventListener("submit", function(e) {
      // Check errors
      checkSelect(problem);

      // If form is invalid, prevent submission
      if (!form.checkValidity()) {
        e.preventDefault();
        alert('Please fix form errors.');
      }
    });
  });
</script>
<title>Select Menus</title>
</head>
<body>
<form id="my-form" method="post" novalidate>
  <label for="problem">50 + 50? *:</label>
  <select name="problem" id="problem" required>
    <option value="0">--Please Choose--</option>
    <option value="50">50</option>
    <option value="100">100</option>
    <option value="250">250</option>
  </select>
```

```
-------Lines 70 through 74 Omitted-------
```

Code Explanation

1. Note that the code listens for a `change` event on the `select` field.
2. Open `FormValidation/Demos/select-menus.html` and submit the form without selecting an answer. You should see an error:



Please choose an answer.

3. As soon as you make a selection, the error should disappear. If you re-select the first option, the error should reappear.

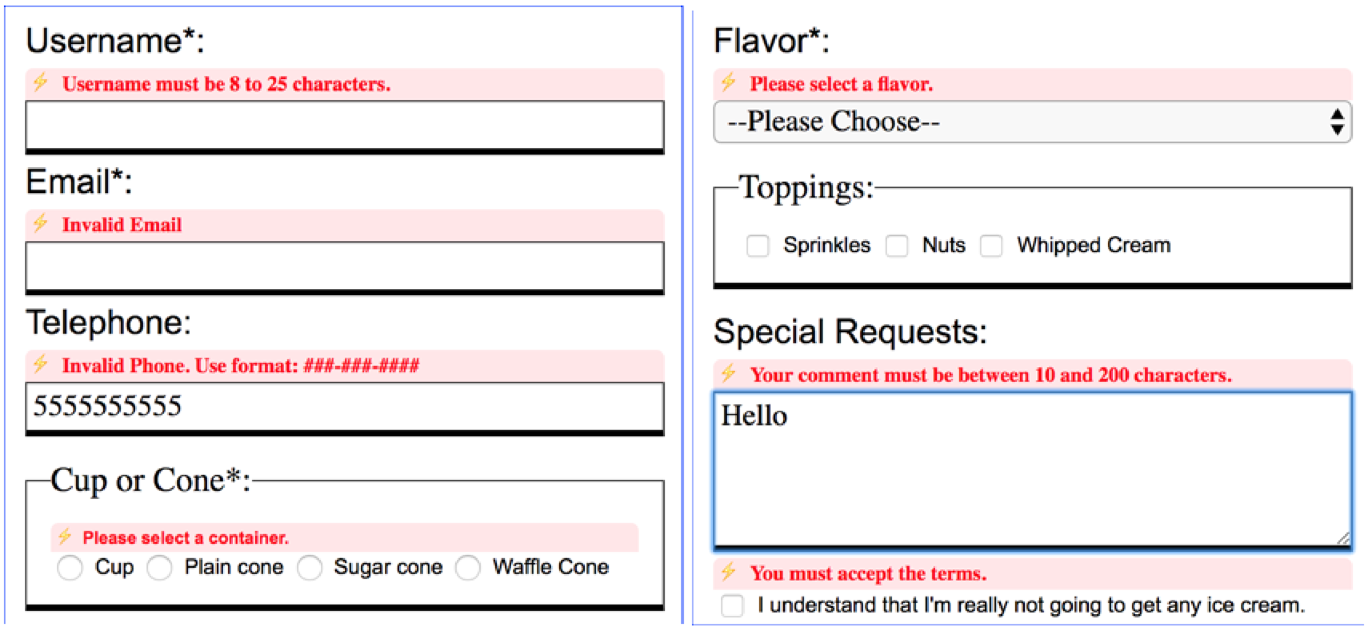
Exercise 5: Validating the Ice Cream Order Form

25 to 40 minutes

In this exercise, you will write JavaScript to validate the ice cream order form we saw at the beginning of this lesson.

1. Open `FormValidation/Exercises/ice-cream.html` for editing.
   A. Add a `script` tag to use `ice-cream.js`.
   B. Turn off HTML form validation.
2. Open `FormValidation/Exercises/ice-cream.js` for editing.
   A. Notice that the following functions are already written:
      i. `addError()`
      ii. `removeError()`
      iii. `checkField()`
      iv. `checkSelect()`
3. Write code to add validation for the following fields:
   A. `email`
   B. `phone`
   C. `username`
   D. `container`
   E. `flavor`
   F. `requests`
   G. `terms`
4. Each of the above fields should be validated when its value is changed. All fields should be validated when the form is submitted. Be sure to prevent the form from submitting if any field is invalid.
5. Test your solution in a browser.

The following screenshot shows the form (split into two parts) after submitting with errors in all fields:



Username*:

⚡ Username must be 8 to 25 characters.

Email*:

⚡ Invalid Email

Telephone:

⚡ Invalid Phone. Use format: ###-###-####

5555555555

Cup or Cone*:

⚡ Please select a container.

○ Cup ○ Plain cone ○ Sugar cone ○ Waffle Cone

Flavor*:

⚡ Please select a flavor.

--Please Choose--

Toppings:

☐ Sprinkles ☐ Nuts ☐ Whipped Cream

Special Requests:

⚡ Your comment must be between 10 and 200 characters.

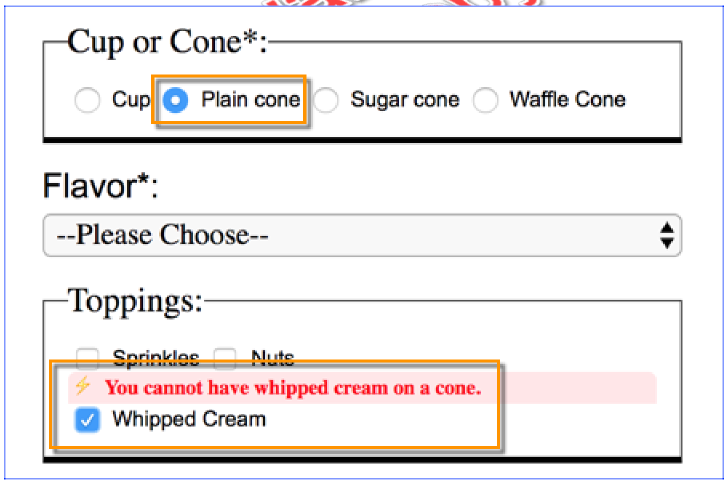
Hello

⚡ You must accept the terms.

☐ I understand that I'm really not going to get any ice cream.

Challenge

Write code to add an error if the user orders whipped cream on a cone:



Cup or Cone*:

○ Cup ● Plain cone ○ Sugar cone ○ Waffle Cone

Flavor*:

--Please Choose--

Toppings:

☐ Sprinkles ☐ Nuts

⚡ You cannot have whipped cream on a cone.

☑ Whipped Cream

Note that you will want to perform this check:

1. When the user clicks the "Whipped Cream" checkbox.
2. When the user selects a container.
3. When the user submits the form.

Solution: FormValidation/Solutions/ice-cream.html

```
-------Lines 1 through 6 Omitted-------
<link rel="stylesheet" href="../styles.css">
<script src="ice-cream.js"></script>
<title>Ice Cream Order Form</title>
</head>
<body>
<form id="ice-cream-form" method="post" novalidate>
-------Lines 13 through 80 Omitted-------
```

Solution: FormValidation/Solutions/ice-cream.js

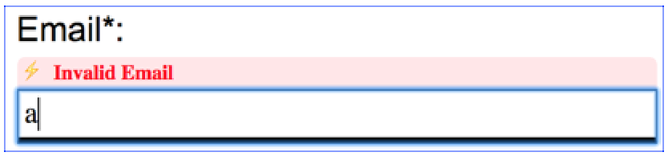
```
-------Lines 1 through 38 Omitted-------
window.addEventListener('load', function(e) {
  const form  = document.getElementById('ice-cream-form');
  const email = form.email;
  email.dataset.errorMsg = 'Invalid Email';
  const phone = form.phone;
  phone.dataset.errorMsg = 'Invalid Phone. Use format: ###-###-####';
  const username = form.username;
  username.dataset.errorMsg = 'Username must be 8 to 25 characters.';
  const container = form.container[0];
  container.dataset.errorMsg = 'Please select a container.';
  const flavor = form.flavor;
  flavor.dataset.errorMsg = 'Please select a flavor.';
  const terms = form.terms;
  terms.dataset.errorMsg = 'You must accept the terms.';
  const requests = form.requests;
  requests.dataset.errorMsg = 'Your comment must be between ' +
    requests.minLength + ' and ' + requests.maxLength +
    ' characters.';

  username.addEventListener("input", function(e) {
    checkField(username);
  });

  email.addEventListener("input", function(e) {
    checkField(email);
  });

  phone.addEventListener("input", function(e) {
    checkField(phone);
  });

  for (radio of form.container) {
    radio.addEventListener("click", function(e) {
      checkField(container);
    });
  }

  flavor.addEventListener("change", function(e) {
    checkSelect(flavor);
  });

  requests.addEventListener("input", function(e) {
    checkField(requests);
  });

  terms.addEventListener("click", function(e) {
    checkField(terms);
  });

  form.addEventListener("submit", function(e) {
    // Check errors
    checkField(username);
    checkField(email);
    checkField(phone);
    checkField(container);
    checkSelect(flavor);
    checkField(requests);
    checkField(terms);

    // If form is invalid, prevent submission
    if (!form.checkValidity()) {
      e.preventDefault();
      alert('Please fix form errors.');
    }
  });

});
```

Challenge Solution: FormValidation/Solutions/ice-cream-challenge.js

```
-------Lines 1 through 38 Omitted-------
function getSelectedRadio(radioArray) {
  for (btn of radioArray) {
    if (btn.checked) {
      return btn;
    }
  }
  return null;
}

function checkWhip(whip) {
  const radioArray = document.querySelectorAll('input[name="container"]');
  const selectedContainer = getSelectedRadio(radioArray);
  if (!selectedContainer) {
    return true; //container not selected
  }
  if (whip.checked && selectedContainer.value !== 'cup') {
    addError(whip);
    return false;
  } else {
    removeError(whip);
  }
  return true;
}

window.addEventListener('load', function(e) {
  const form  = document.getElementById('ice-cream-form');
  const email = form.email;
  email.dataset.errorMsg = 'Invalid Email';
  const phone = form.phone;
  phone.dataset.errorMsg = 'Invalid Phone. Use format: ###-###-####';
  const username = form.username;
  username.dataset.errorMsg = 'Username must be 8 to 25 characters.';
  const container = form.container[0];
  container.dataset.errorMsg = 'Please select a container.';
  const flavor = form.flavor;
  flavor.dataset.errorMsg = 'Please select a flavor.';
  const terms = form.terms;
  terms.dataset.errorMsg = 'You must accept the terms.';
  const requests = form.requests;
  requests.dataset.errorMsg = 'Your comment must be between ' +
    requests.minLength + ' and ' + requests.maxLength +
    ' characters.';
  const whip = form.whip;
  whip.dataset.errorMsg = 'You cannot have whipped cream on a cone.';

  username.addEventListener("input", function(e) {
    checkField(username);
  });

  email.addEventListener("input", function(e) {
    checkField(email);
  });

  phone.addEventListener("input", function(e) {
    checkField(phone);
  });

  for (radio of form.container) {
    radio.addEventListener("click", function(e) {
      checkField(container);
      checkWhip(whip);
    });
  }

  flavor.addEventListener("change", function(e) {
    checkSelect(flavor);
  });

  requests.addEventListener("input", function(e) {
    checkField(requests);
  });

  terms.addEventListener("click", function(e) {
    checkField(terms);
  });

  whip.addEventListener("click", function(e) {
    checkWhip(whip);
  });

  form.addEventListener("submit", function(e) {
    // Check errors
    checkField(username);
    checkField(email);
    checkField(phone);
    checkField(container);
    checkSelect(flavor);
    checkField(requests);
    checkField(terms);
    const whipValid = checkWhip(whip);

    // If form is invalid, prevent submission
    if (!form.checkValidity() || !whipValid) {
      e.preventDefault();
      alert('Please fix form errors.');
    }
  });

});
```

2.11. Giving the User a Chance

While it is great to capture errors early, so the user can fix them right away, some users might be a little put off by errors that show up before they have had a chance to finish an individual entry. For example, the way our ice cream form works now, as soon as the user starts entering an email address, an error shows up:



One solution would be to listen for `change` events, which do not fire until the field loses focus, instead of `input` events, which fire every time a character is added or removed. This solution works well the first time the user enters data, but it is not as nice when the user is correcting a field that already has an error as in the following scenario:

1. The user enters an invalid email address and moves on to the next field, causing an error to show up.
2. The user returns to the email field and fixes the email address. The error will not go away until the `change` event fires, which won't occur until the user leaves the field.

In the above scenario, it would be nicer to hide the error as soon as the field is valid and then show it again if the field becomes invalid again. We can do this by keeping track of which fields have been "touched." We will consider a field to have been touched if the user changes its value or if the user has submitted the form and received validation errors.

Note that this will only apply to text-like `input` and `textarea` fields as we use `change` and `click` events to initiate validation on other field types. We can get all `input` and `textarea` fields using:

```
const inputFields = document.querySelectorAll('input, textarea');
```

Take a look at the following code:

Demo 2.12: FormValidation/Demos/ice-cream-final.js

```
-------Lines 1 through 81 Omitted-------
// Get the input and textarea fields
const inputFields = document.querySelectorAll('input, textarea');

// Loop through the input fields, marking them all "untouched"
for (field of inputFields) {
  field.dataset.status = 'untouched';
}

// When a user changes the value of a text-like input or textarea,
//  mark the field "touched"
//  validate the field
username.addEventListener("change", function(e) {
  username.dataset.status = 'touched';
  checkField(username);
});

// When a user inputs data into a text-like input or textarea
//  that has been touched, validate the field
username.addEventListener("input", function(e) {
  if (username.dataset.status === 'touched') {
    checkField(username);
  }
});

email.addEventListener("change", function(e) {
  email.dataset.status = 'touched';
  checkField(email);
});
email.addEventListener("input", function(e) {
  if (email.dataset.status === 'touched') {
    checkField(email);
  }
});

phone.addEventListener("change", function(e) {
  phone.dataset.status = 'touched';
  checkField(phone);
});
phone.addEventListener("input", function(e) {
  if (phone.dataset.status === 'touched') {
    checkField(phone);
  }
});
```

```
  requests.addEventListener("change", function(e) {
    requests.dataset.status = 'touched';
    checkField(requests);
  });
  requests.addEventListener("input", function(e) {
    if (requests.dataset.status === 'touched') {
      checkField(requests);
    }
  });
-------Lines 135 through 155 Omitted-------
    // Mark all fields touched
    for (field of inputFields) {
      field.dataset.status = 'touched';
    }
-------Lines 160 through 177 Omitted-------
```

Code Explanation

The following code will not validate an `input` or `textarea` field on each input unless the user has previously changed that field or submitted the form. **Things to note:**

1. We save all the `input` and `textarea` fields as a `NodeList` (similar to an array of elements) called `inputFields`.
2. We loop through the input fields, setting the `dataset.status` property of each field to "untouched."
3. We add `change` event listeners to each text-like `input` and the "requests" `textarea`. When the `change` event occurs, we set the `dataset.status` property of the field to "touched" and we validate the field.
4. We modified the `input` event listeners to check if the `dataset.status` property is "touched", and if it is, we validate the field.
5. We added a `for` loop in the callback function of the `submit` event listener to set the `dataset.status` property of all the `input` and `textarea` fields to "touched".

Open `FormValidation/Demos/ice-cream-final.html` in your browser to try it out.

Conclusion

In this lesson, you have learned to write clean, reusable form validation scripts.

LESSON 3
Styling Forms with CSS

Topics Covered

☑ Form layout with CSS.

☑ Validation-related pseudo-classes.

Introduction

Styling form controls with CSS is a bit tricky. Browsers differ slightly in how they format different form fields and users get accustomed to how forms are displayed by their browser of choice. If your form fields look different from the other forms they have used on the web, you risk confusing them. In this lesson, you will learn how you can use CSS to style forms and to use CSS pseudo-classes to provide visual feedback when form fields are invalid.

3.1. General Form Layout

There are many ways to lay out forms. One common practice is to separate entries using `<div>` tags, as in the following example:

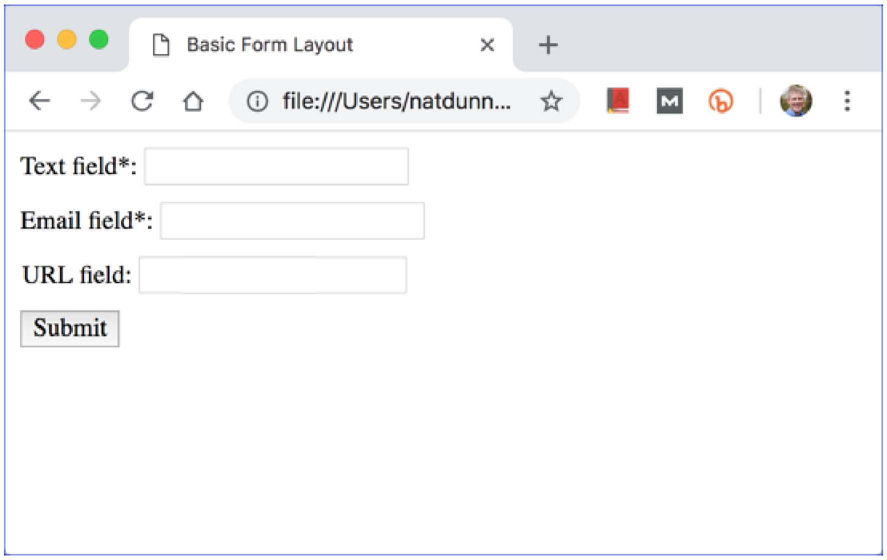
Demo 3.1: CssForms/Demos/form-start.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1.0">
<link rel="stylesheet" href="../normalize.css">
<style>
  div {
    margin: 10px;
  }
</style>
<title>Basic Form Layout</title>
</head>
<body>
<form id="my-form" method="post">
  <div>
    <label for="textfield">Text field*:</label>
    <input type="text" id="textfield" name="textfield"
    required minlength="5" maxlength="10">
  </div>
  <div>
    <label for="emailfield">Email field*:</label>
    <input type="email" id="emailfield" name="emailfield" required>
  </div>
  <div>
    <label for="urlfield">URL field:</label>
    <input type="url" id="urlfield" name="urlfield" pattern="https:.*">
  </div>
  <div>
    <button>Submit</button>
  </div>
</form>
</body>
</html>
```

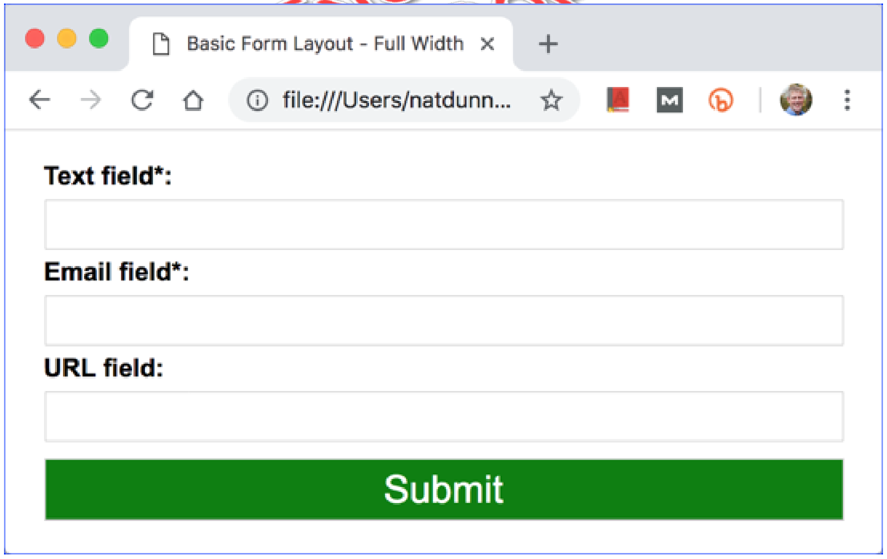
Code Explanation

Note that we have used `normalize.css` to normalize styles across browsers.[6] Except for adding some margin to the `div` elements, the form is unstyled. It appears as follows in Google Chrome:

6. For details on `normalize.css` and to get the latest version, see `https://necolas.github.io/normalize.css/`.



One common way to style forms is to set most form controls to extend the full width of the form with the `label`s above the controls, like this:



Here is the CSS we used for this form:

Demo 3.2: CssForms/Demos/layout.css

```
form {
  width: 500px;
  margin: 20px auto;
}

label {
  display: block;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: bold;
  margin: 5px 0px;
}

input {
  box-sizing: border-box;
  font-size: large;
  padding: 4px;
  width: 100%;
}

button {
  background-color: green;
  color: white;
  font-family: Arial, Helvetica, sans-serif;
  font-size: x-large;
  margin-top: 10px;
  padding: 5px;
  width: 100%;
}
```

Code Explanation

Some of what we have done:

1. Added a `width` and `margin` to the form.
2. Set the `label` elements to blocks and added some `margin`.
3. Set the `width` of the `input` elements to `100%` and set the `box-sizing` property to `border-box` so that the `input` boxes take up the full width (and only the full width) of the form.
4. Set the `width` of the `button` element to `100%` and added some styling.

Because we have made the `labels` block-level elements, we no longer need to use `div` elements to visually separate the fields. So, we are able to simplify our HTML to:

Demo 3.3: CssForms/Demos/form-layout.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1.0">
<link rel="stylesheet" href="../normalize.css">
<link rel="stylesheet" href="layout.css">
<title>Basic Form Layout - Full Width</title>
</head>
<body>
<form id="my-form" method="post">
 <label for="textfield">Text field*:</label>
 <input type="text" id="textfield" name="textfield"
  required minlength="5" maxlength="10">
 <label for="emailfield">Email field*:</label>
 <input type="email" id="emailfield" name="emailfield" required>
 <label for="urlfield">URL field:</label>
 <input type="url" id="urlfield" name="urlfield" pattern="https:.*">

 <button>Submit</button>
</form>
</body>
</html>
```

3.2. Form-field Pseudo-Classes

Next, we will add some visual validation to show when form fields are invalid. But before doing that, let’s look at the pseudo-classes that are available for form fields.

Form-field Pseudo-classes

Pseudo-class	Represents	Can be applied to
`:checked`	A selected or checked option.	Radio buttons, checkboxes and options in select menus.
`:default`	The default element among a group of elements.	Pre-selected radio buttons, checkboxes, and options in select menus.
`:disabled`	A disabled element.	Any form element.
`:enabled`	An enabled element.	Any form element.
`:focus`	An element that has focus (has been clicked or tapped or tabbed into).	Any form element.
`:indeterminate`	A form element with an indeterminate state.	Although not solely, this is mostly used (and rarely at that) to indicate that no radio button of a group is selected.
`:in-range`	An input element with a value in between its `min` and `max` properties.	Any input element that has `min` and `max` properties set.
`:invalid`	A form element with an invalid value.	Any form element that can be validated.
`:optional`	A form element that is not required.	Any form element that can be required.
`:out-of-range`	An input element with a value outside of its `min` and `max` properties.	Any input element that has `min` and `max` properties set.
`:required`	A form element that is required.	Any form element that can be required.
`:valid`	A form element with a valid value.	Any form element that can be validated.

Several of the above pseudo-classes are rarely used for several reasons:

1. **Conveying meaning**. It can be difficult to convey meaning to form fields with CSS alone. For example, how would you style an element to indicate to the user that the number entered is in or out of range?
2. **Timing of error**. The CSS rules you create will take effect immediately. For example, if you decide to give invalid `input` fields a pink background, all required `input` fields will be pink as soon as the page loads, before the user has had a chance to enter any data. Ideally, users wouldn't see errors until after they attempted to fill out a field. The same is true for `input` fields that must match a certain pattern or be a certain length. As soon as a user starts entering text, the background color will change to pink and remain so until the user enters valid data. Again, it would be better to let the user finish entering the data before indicating that the field is invalid.

3. **Consistency**. It is difficult to apply styles to text fields, checkboxes, radio buttons, select menus and textareas that are consistent. For example, you can give text-like `input` fields a background color and even a background image to indicate that they are invalid, but you cannot do the same thing with checkboxes, radio buttons and select menus.

Because of the above issues, we recommend relying more on JavaScript than CSS to provide feedback to the user on the validity of form fields. That said, we will show how you can apply some of these pseudo-classes as you may come across cases in which they come in handy.

3.3. Applying Pseudo-Classes to Forms

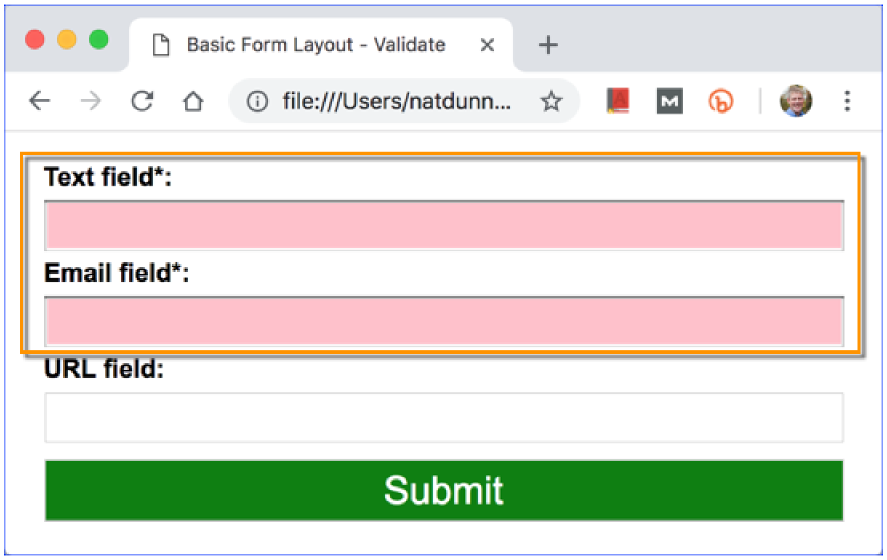
In the following demo, we begin to apply pseudo-classes to form fields.

Demo 3.4: CssForms/Demos/pseudo-classes1.css

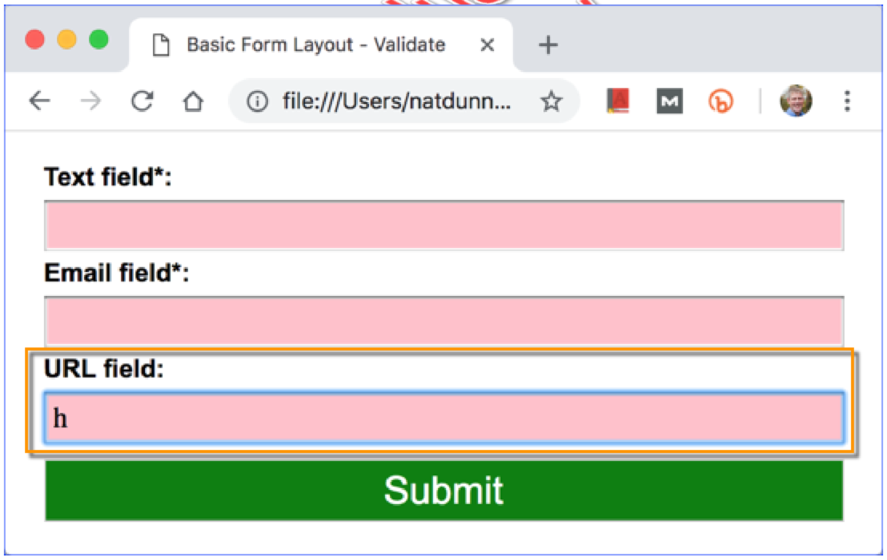
```css
input:invalid {
  background-color: pink;
}
```

Code Explanation

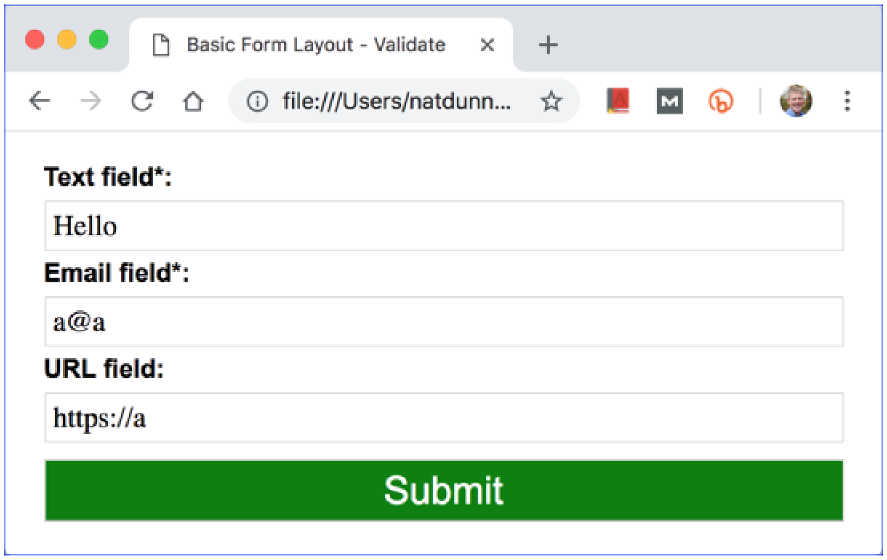
1. We start with just adding a pink background to invalid `input` elements.
2. Open `CssForms/Demos/form-validate.html`, which includes the `pseudo-classes1.css` stylesheet, in your browser. You'll see that as soon as the page loads the two required fields show up with pink backgrounds:



3. As soon as you begin to enter a URL, that field will get a pink background as well:



4. The pink backgrounds will go away as soon as the fields meet the minimum validation requirements:



Text field*:

Hello

Email field*:

a@a

URL field:

https://a

Submit

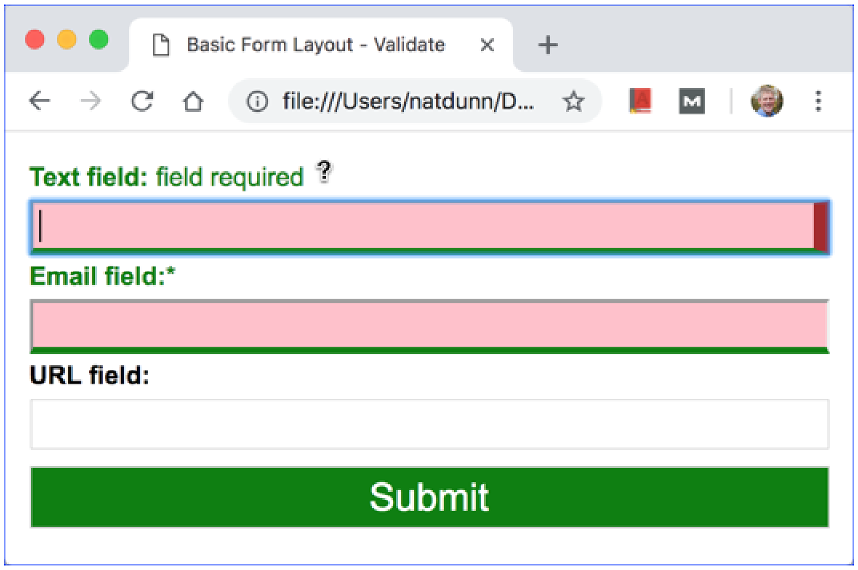
Now let's add styling to fields that are required and fields that have focus. Here's our modified CSS:

Demo 3.5: CssForms/Demos/pseudo-classes2.css

```css
input:invalid {
  background-color: pink;
}

input:required {
  border-bottom: 4px solid green;
}

input:focus {
  border-right: 10px solid brown;
}

label.required {
  color: green;
  cursor: help;
}

label.required::after {
  color: green;
  content: "*";
  font-weight: normal;
}

label.required:hover::after {
  content: "field required";
  margin-left: 5px;
}
```

Code Explanation

The screenshot below shows our form when the first element has focus and the mouse is hovering over the “Text field” `label`:


Basic Form Layout - Validate
file:///Users/natdunn/D...
Text field: field required ?
Email field:*
URL field:
Submit

1. Open `CssForms/Demos/form-validate.html` in your editor and edit it so that it uses `pseudo-classes2.css` instead of `pseudo-classes1.css`.
2. Open `CssForms/Demos/form-validate.html` in your editor:
   A. Remove the asterisks from the `label`s of the required fields.
   B. Give those same `label` tags the "required" class.

3. Now open `CssForms/Demos/form-validate.html` in your browser.
4. Here is what we have done with form-related pseudo-classes:
   A. Added a solid, green, 4-pixel bottom border to **required** `input` fields.
   B. Added a solid, brown, 10-pixel right border to `input` fields that have focus.
5. In addition, we have:
   A. Added a CSS rule to color required `label`s green and make the cursor a question mark when the `label` is hovered over.
   B. Used the `::after` pseudo-element to add an asterisk to `label`s with the "required" class.
   C. Used the `:hover` pseudo-class along with the `::after` pseudo-element to change the text at the end of a `label` with the "required" class to "field required" when the user hovers over the `label`.

3.4. Radio Buttons, Checkboxes, and Fieldsets

Now let’s see how we can handle radio buttons and checkboxes. Let’s start with the HTML:

Demo 3.6: CssForms/Demos/radios-checkboxes.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1.0">
<link rel="stylesheet" href="../normalize.css">
<link rel="stylesheet" href="layout.css">
<link rel="stylesheet" href="pseudo-classes2.css">
<link rel="stylesheet" href="radios-checkboxes.css">
<title>Radio buttons and Checkboxes</title>
</head>
<body>
<form>
  <fieldset>
    <legend class="required">Question:</legend>
    <label>
      <input type="radio" name="answer" value="1" required> A
    </label>
    <label>
      <input type="radio" name="answer" value="2" required> B
    </label>
    <label>
      <input type="radio" name="answer" value="3" required> C
    </label>
    <label>
      <input type="radio" name="answer" value="4" required> D
    </label>
  </fieldset>

  <input type="checkbox" name="terms" id="terms" required>
  <label for="terms" class="required">
    I understand that it's important
    that I check this box.
  </label>

  <button>Submit</button>
</form>
</body>
</html>
```

Code Explanation

Notice that the form contains a group of required radio buttons in a `fieldset` and a required `checkbox` with a trailing `label`.

Also notice that, in addition to the CSS files we included earlier, we have added a new `radios-checkboxes.css` CSS file.

Now let's look at the new CSS file that we have added:

Demo 3.7: CssForms/Demos/radios-checkboxes.css

```css
fieldset {
  margin: 10px 0px;
  padding: 15px;
}

legend {
  font-weight: bold;
}

/* Undo block-level formatting for the fieldset label
  and the label for the terms checkbox */
fieldset label,
label[for='terms'] {
  display: inline;
  font-weight: normal;
}

/* Radio buttons and checkboxes shouldn't have 100% width */
input[type='checkbox'],
input[type='radio'] {
  margin: 0 5px;
  width: auto;
}

input[type='checkbox']:invalid + label {
  color: red;
}

legend.required {
  color: green;
  cursor: help;
}

legend.required::after  {
  color: green;
  content: "*";
  font-weight: normal;
}

legend.required:hover::after  {
  content: "field required";
  margin-left: 5px;
}
```

Code Explanation

Much of this code should be clear. Here's what we've done:

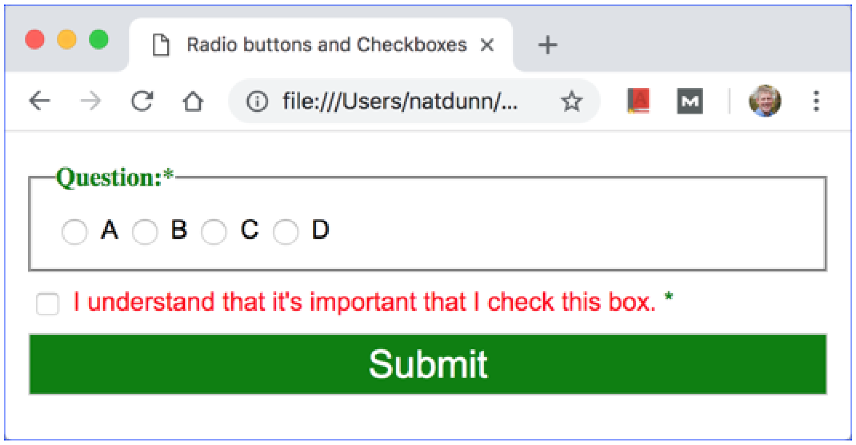
1. Added margin and padding to `fieldsets`.
2. Made `legends` bold.
3. Changed the font weight to normal and display to inline of the `labels` for checkboxes and radio buttons.
4. Removed the 100% width from checkboxes and radio buttons. That 100% width was added to all `inputs` in `layout.css`.
5. Set the color of a `label` that follows an invalid checkbox to red:

   ```
   input[type='checkbox']:invalid + label {
     color: red;
   }
   ```

   This uses the *adjacent sibling combinator*, documented at `https://developer.mozilla.org/en-US/docs/Web/CSS/Next-sibling_combinator`.
6. Set `legends` that have the "required" class to match `labels` that have the required class as set in `pseudo-classes2.css`. Note that these could be combined into one rule as could the two following rules that add content after the `legend`:

   ```
   label.required, legend.required {
     color: green;
     cursor: help;
   }

   label.required::after, legend.required::after  {
     content: "*";
     color: green;
     font-weight: normal;
   }

   label.required:hover::after, legend.required:hover::after  {
     content: "field required";
     margin-left: 5px;
   }
   ```

The screenshot below shows our form when the page first loads:

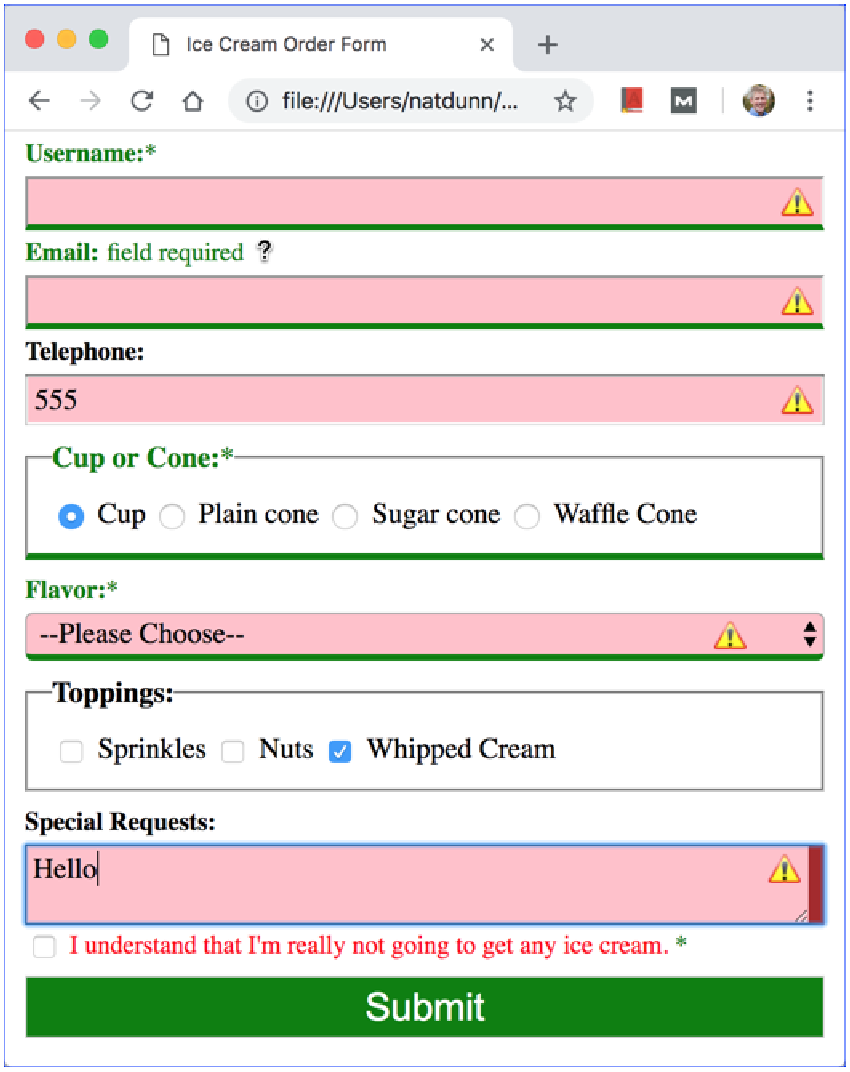

Radio buttons and Checkboxes
file:///Users/natdunn/...
Question:*
A
B
C
D
I understand that it's important that I check this box. *
Submit

1. Open `CssForms/Demos/radio-checkboxes.html` in your browser.
2. Hover over either of the `labels` to see the text change.
3. Check the checkbox to see its label change from red to green.

Exercise 6: Styling Forms

30 to 60 minutes

We end this lesson with an open-ended exercise to give you practice styling forms. You can either try to make the form look and act like the one below or style the form however you like. You can open `CssForms/Solutions/ice-cream.html` in your browser to see how it works when users interact with it.



Username:*

Email: field required ?

Telephone:

555

Cup or Cone:*

● Cup ○ Plain cone ○ Sugar cone ○ Waffle Cone

Flavor:*

--Please Choose--

Toppings:

☐ Sprinkles ☐ Nuts ☑ Whipped Cream

Special Requests:

Hello

☐ I understand that I'm really not going to get any ice cream. *

Submit

1. Open `CssForms/Exercises/ice-cream.html` in your editor and review the code.
2. Open `CssForms/Exercises/ice-cream.css` in your editor. It is currently empty. Use it to style the ice cream order form.

3. Note that there are a couple of images in `CssForms/Images/` that you may wish to use.

Solution: CssForms/Solutions/ice-cream.css

```css
form {
  margin: auto;
  width: 500px;
}

label {
  display: block;
  font-weight: bold;
  margin: 5px 0px;
}

legend {
  font-weight: bold;
}

input,
select,
textarea,
fieldset {
  box-sizing: border-box;
  font-size: large;
  padding: 4px;
  width: 100%;
}

fieldset {
  margin: 10px 0px;
  padding: 15px;
}

/* Undo block-level formatting for the fieldset label
  and the label for the terms checkbox */
fieldset label,
label[for='terms'] {
  display: inline;
  font-weight: normal;
}

/* Radio buttons and checkboxes shouldn't have 100% width */
input[type='checkbox'],
input[type='radio'] {
  margin: 0 5px;
  width: auto;
}
```

```

button {
  background-color: green;
  color: white;
  font-family: Arial, Helvetica, sans-serif;
  font-size: x-large;
  margin-top: 10px;
  padding: 5px;
  width: 100%;
}

/* Begin validation-related pseudo-classes */
input:invalid,
textarea:invalid,
select:invalid {
  background-color: pink;
  background-image: url('../Images/warning.png');
  background-position: 99% 5px;
  background-repeat: no-repeat;
}

select:invalid {
  background-position: 90%;
}

input:required,
select:required,
#container {
  border-bottom: 4px solid green;
}

input:focus,
select:focus,
textarea:focus {
  border-right: 10px solid brown;
}

label.required,
legend.required {
  color: green;
  cursor: help;
}

label.required::after,
legend.required::after  {
  color: green;
  content: "*";
  font-weight: normal;
}

label.required:hover::after,
legend.required:hover::after  {
  content: "field required";
  margin-left: 5px;
}

input[type='checkbox']:invalid + label {
  color: red;
}
```

Conclusion

In this lesson, you have learned to use CSS to style forms.

LESSON 4
Regular Expressions

Topics Covered

☑ Regular expressions for advanced form validation.

☑ Regular expressions and backreferences to clean up form entries.

Introduction

Regular expressions are used to do sophisticated pattern matching, which can often be helpful in form validation. For example, a regular expression can be used to check whether an email address entered into a form field is syntactically correct. JavaScript supports Perl-compatible regular expressions.

4.1. Getting Started

There are two ways to create a regular expression in JavaScript:

1. Using literal syntax

   ```
   const reExample = /pattern/;
   ```

2. Using the `RegExp()` constructor

   ```
   const reExample = new RegExp("pattern");
   ```

There is no difference between the two.

For example, we could create a regular expression for a social security number like this:

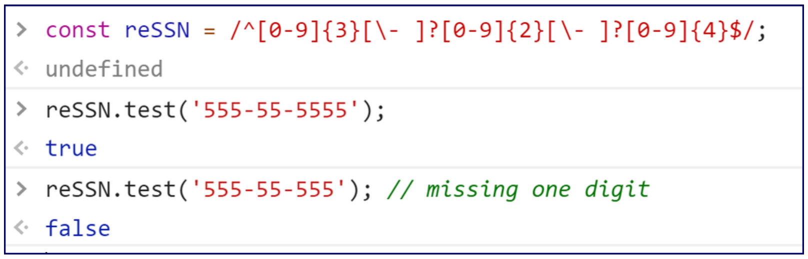
```
// literal syntax
const reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;

// RegExp() contructor syntax
const reSSN = new RegExp("^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$");
```

❖ 4.1.1. JavaScript's Regular Expression test() Method

The `test()` method takes one argument, a string, and checks whether that string contains a match of the pattern specified by the regular expression. It returns `true` if it does contain a match and `false` if it does not. This method is very useful in form validation scripts. The following code sample shows how it can be used for checking a social security number. Don't worry about the syntax of the regular expression itself. We'll cover that shortly.

```
const reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;
reSSN.test('555-55-5555');
reSSN.test('555-55-555'); // missing one digit
```

Here we show this code run in Chrome DevTools Console:



```
> const reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;
<· undefined
> reSSN.test('555-55-5555');
<· true
> reSSN.test('555-55-555'); // missing one digit
<· false
```

4.2. Regular Expression Syntax

A regular expression is a pattern that specifies a list of characters. In this section, we will look at how those characters are specified.

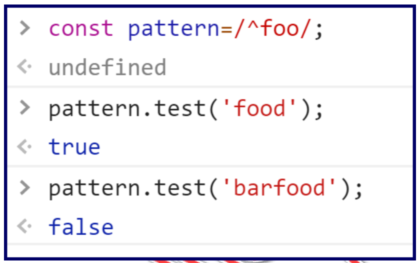
Try These!

We encourage you to try these regular expressions in Google Chrome's console.

❖ 4.2.1. Start and End (^ $)

A caret (`^`) at the beginning of a regular expression indicates that the string being searched must start with this pattern.

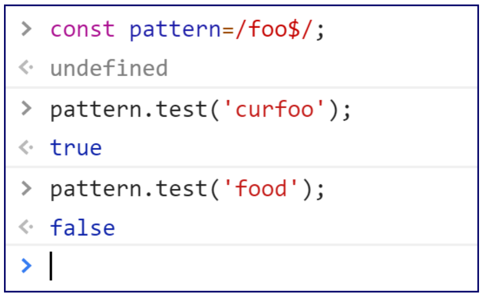
- The pattern `^foo` can be found in "food", but not in "barfood".



```
> const pattern=/^foo/;
<· undefined
> pattern.test('food');
<· true
> pattern.test('barfood');
<· false
```

A dollar sign (`$`) at the end of a regular expression indicates that the string being searched must end with this pattern.

- The pattern `foo$` can be found in "curfoo", but not in "food".



```
> const pattern=/foo$/;
<· undefined
> pattern.test('curfoo');
<· true
> pattern.test('food');
<· false
> |
```

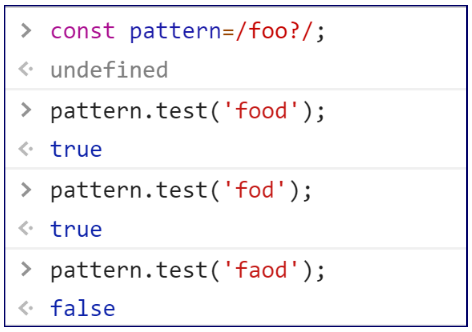
❖ 4.2.2. Number of Occurrences (? + * {})

The following symbols affect the number of occurrences of the preceding character[7]: `?`, `+`, `*`, and `{}`.

7. Or characters if parentheses are used.

A question mark (?) indicates that the preceding character should appear zero or one times in the pattern.

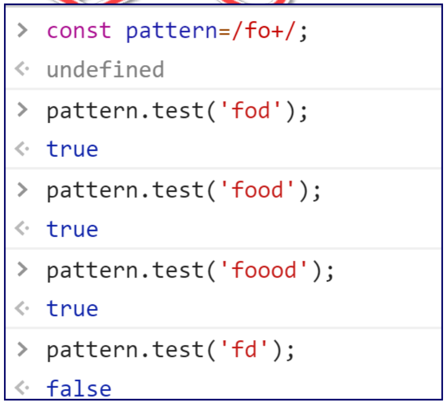
- The pattern `foo?` can be found in “food” and “fod”, but not “faod”.



```
> const pattern=/foo?/;
<· undefined
> pattern.test('food');
<· true
> pattern.test('fod');
<· true
> pattern.test('faod');
<· false
```

A plus sign (+) indicates that the preceding character should appear one or more times in the pattern.

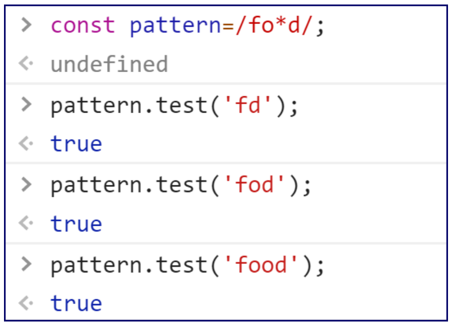
- The pattern `fo+` can be found in “fod”, “food” and “foood”, but not “fd”.



```
> const pattern=/fo+/;
<· undefined
> pattern.test('fod');
<· true
> pattern.test('food');
<· true
> pattern.test('foood');
<· true
> pattern.test('fd');
<· false
```

An asterisk (*) indicates that the preceding character should appear zero or more times in the pattern.

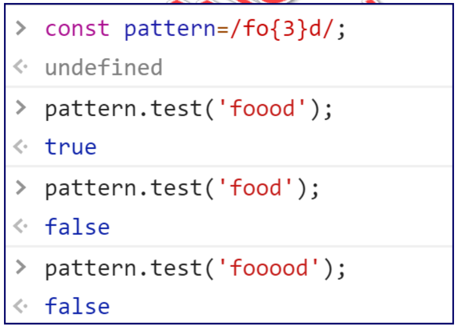
- The pattern `fo*d` can be found in “fd”, “fod” and “food”.



```
> const pattern=/fo*d/;
<· undefined
> pattern.test('fd');
<· true
> pattern.test('fod');
<· true
> pattern.test('food');
<· true
```

Curly brackets with one parameter (`{n}`) indicate that the preceding character should appear exactly `n` times in the pattern.

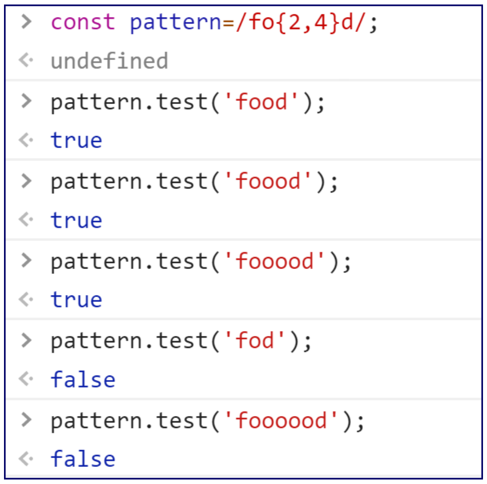
- The pattern `fo{3}d` can be found in "foood", but not "food" or "fooood".



```
> const pattern=/fo{3}d/;
<· undefined
> pattern.test('foood');
<· true
> pattern.test('food');
<· false
> pattern.test('fooood');
<· false
```

Curly brackets with two parameters (`{n1, n2}`) indicate that the preceding character should appear between `n1` and `n2` times in the pattern.

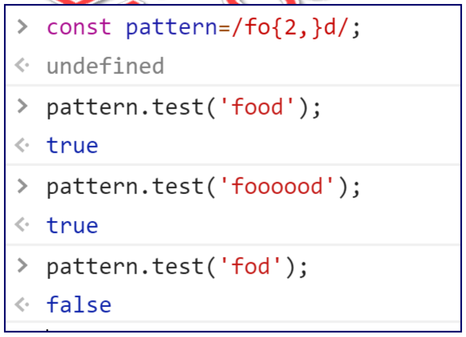
- The pattern `fo{2,4}d` can be found in "food", "foood", and "fooood", but not "fod" or "foooood".



```
> const pattern=/fo{2,4}d/;
<· undefined
> pattern.test('food');
<· true
> pattern.test('foood');
<· true
> pattern.test('fooood');
<· true
> pattern.test('fod');
<· false
> pattern.test('foooood');
<· false
```

Curly brackets with one parameter and an empty second paramenter (`{n,}`) indicate that the preceding character should appear at least `n` times in the pattern.

- The pattern `fo{2,}d` can be found in "food" and "foooood", but not "fod".

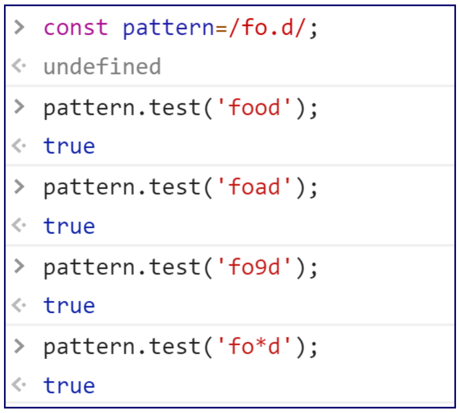


```
> const pattern=/fo{2,}d/;
<· undefined
> pattern.test('food');
<· true
> pattern.test('foooood');
<· true
> pattern.test('fod');
<· false
```

❖ 4.2.3. Common Characters (. \d \D \w \W \s \S)

A period (`.`) represents any character except a newline.

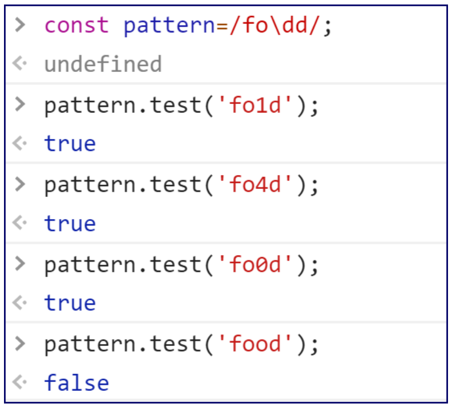
- The pattern `fo.d` can be found in "food", "foad", "fo9d", and "fo*d".



```
> const pattern=/fo.d/;
<· undefined
> pattern.test('food');
<· true
> pattern.test('foad');
<· true
> pattern.test('fo9d');
<· true
> pattern.test('fo*d');
<· true
```

Backslash-d (`\d`) represents any digit. It is the equivalent of [0-9] (see page 113).

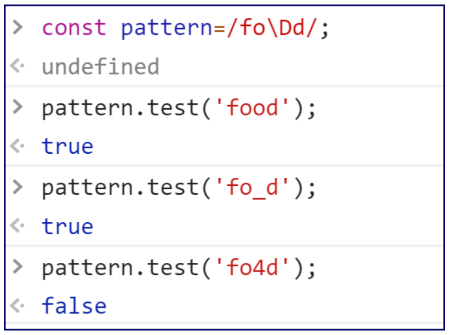
- The pattern `fo\dd` can be found in "fo1d", "fo4d" and "fo0d", but not in "food".



```
> const pattern=/fo\dd/;
<· undefined
> pattern.test('fo1d');
<· true
> pattern.test('fo4d');
<· true
> pattern.test('fo0d');
<· true
> pattern.test('food');
<· false
```

Backslash-D (\D) represents any character **except a digit**. It is the equivalent of [^0-9] (see page 114).

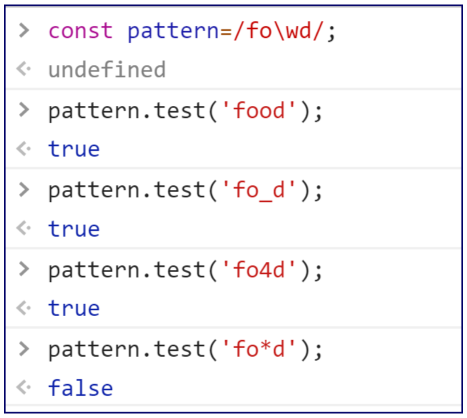
- The pattern `fo\Dd` can be found in "food" and "fo_d", but not in "fo4d".



```
> const pattern=/fo\Dd/;
<· undefined
> pattern.test('food');
<· true
> pattern.test('fo_d');
<· true
> pattern.test('fo4d');
<· false
```

Backslash-w (`\w`) represents any word character (letters, digits, and the underscore (_)).

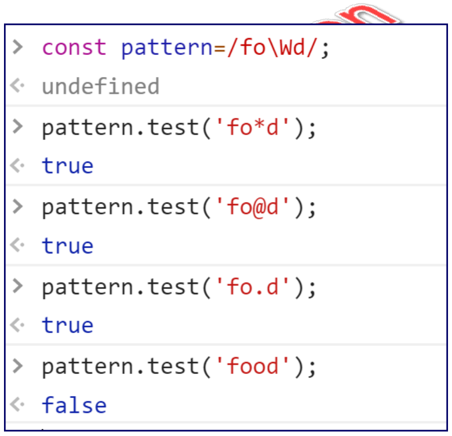
- The pattern `fo\wd` can be found in “food”, “fo_d” and “fo4d”, but not in “fo*d”.



```
> const pattern=/fo\wd/;
<· undefined
> pattern.test('food');
<· true
> pattern.test('fo_d');
<· true
> pattern.test('fo4d');
<· true
> pattern.test('fo*d');
<· false
```

Backslash-W (`\W`) represents any character **except a word character**.

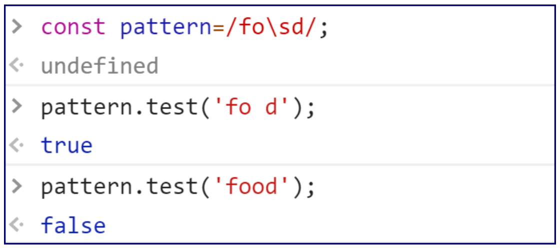
- The pattern `fo\Wd` can be found in “fo*d”, “fo@d” and “fo.d”, but not in “food”.



```
> const pattern=/fo\Wd/;
<· undefined
> pattern.test('fo*d');
<· true
> pattern.test('fo@d');
<· true
> pattern.test('fo.d');
<· true
> pattern.test('food');
<· false
```

Backslash-s (`\s`) represents any whitespace character (e.g, space, tab, newline, etc.).

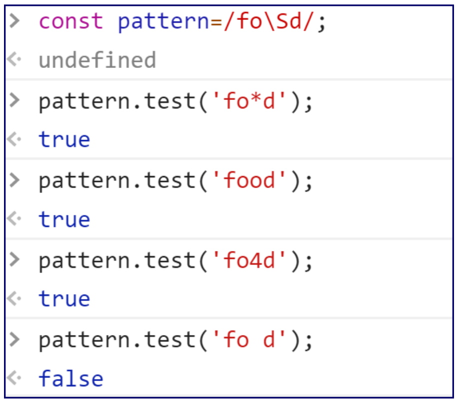
- The pattern `fo\sd` can be found in “fo d”, but not in “food”.



```
> const pattern=/fo\sd/;
<· undefined
> pattern.test('fo d');
<· true
> pattern.test('food');
<· false
```

Backslash-S (`\S`) represents any character except a whitespace character.

- The pattern `fo\Sd` can be found in "fo*d", "food" and "fo4d", but it cannot be found in "fo d".

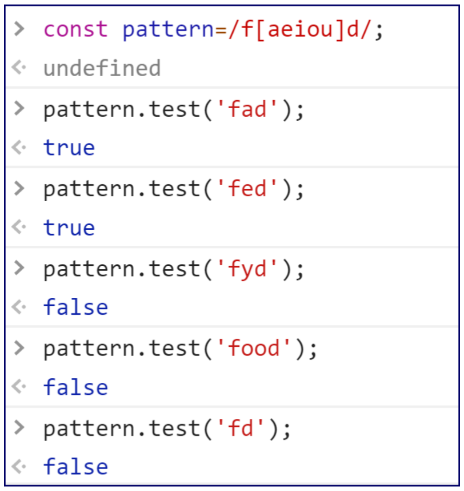


```
> const pattern=/fo\Sd/;
<· undefined
> pattern.test('fo*d');
<· true
> pattern.test('food');
<· true
> pattern.test('fo4d');
<· true
> pattern.test('fo d');
<· false
```

❖ 4.2.4. Grouping ([])

Square brackets (`[ ]`) are used to group options.

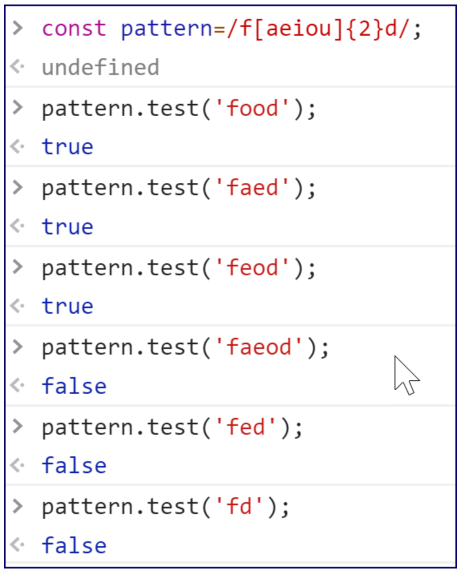
- `[aeiou]` matches an "a", an "e", an "i", an "o", or a "u".
    - The pattern `f[aeiou]d` can be found in "fad" and "fed", but not in "fyd", "food" or "fd".



```
> const pattern=/f[aeiou]d/;
<· undefined
> pattern.test('fad');
<· true
> pattern.test('fed');
<· true
> pattern.test('fyd');
<· false
> pattern.test('food');
<· false
> pattern.test('fd');
<· false
```

- Number of occurrence characters can be used with groups.

- The pattern `f[aeiou]{2}d` can be found in "f**oo**d", "f**ae**d" and "f**eo**d", but not in "faeod", "fed" or "fd".



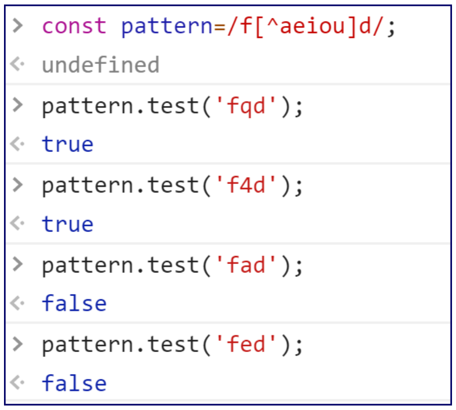
```
> const pattern=/f[aeiou]{2}d/;
<· undefined
> pattern.test('food');
<· true
> pattern.test('faed');
<· true
> pattern.test('feod');
<· true
> pattern.test('faeod');
<· false
> pattern.test('fed');
<· false
> pattern.test('fd');
<· false
```

- Ranges can be created using a dash:
  - `[A-Z]` matches any upper case letter.
  - `[a-z]` matches any lower case letter.
  - `[A-Za-z]` matches any letter, regardless if it is lower case or upper case.

❖ 4.2.5. Negation (^)

When used as the first character within square brackets, the caret (`^`) is used for negation.

- The pattern `f[^aeiou]d` can be found in "fqd" and "f4d", but not in "fad" or "fed".

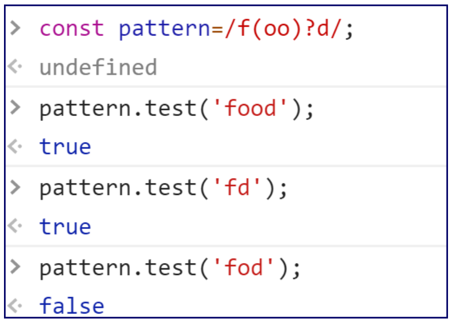


```
> const pattern=/f[^aeiou]d/;
<· undefined
> pattern.test('fqd');
<· true
> pattern.test('f4d');
<· true
> pattern.test('fad');
<· false
> pattern.test('fed');
<· false
```

❖ 4.2.6. Subpatterns (())

Parentheses (()) are used to capture subpatterns.

- The pattern `f(oo)?d` indicates that the subpattern `oo` can show up zero or one time. The pattern can be found in "food" and "fd", but not in "fod".

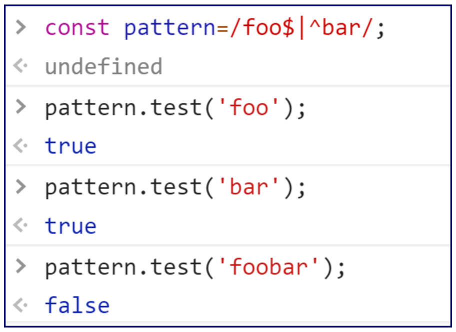


```
> const pattern=/f(oo)?d/;
<· undefined
> pattern.test('food');
<· true
> pattern.test('fd');
<· true
> pattern.test('fod');
<· false
```

❖ 4.2.7. Alternatives (|)

The pipe (|) is used to create optional patterns.

- The pattern `foo$|^bar` can be found in "foo" and "bar", but not "foobar".

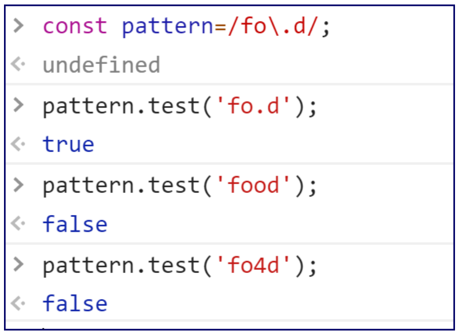


```
> const pattern=/foo$|^bar/;
<· undefined
> pattern.test('foo');
<· true
> pattern.test('bar');
<· true
> pattern.test('foobar');
<· false
```

❖ 4.2.8. Escape Character (\ )

The backslash (\ ) is used to escape special characters, such as periods (`.`), dashes (`-`), forward slashes (`/`) and backslashes (`\`).

- The pattern `fo\.d` can be found in "fo.d", but not in "food" or "fo4d".



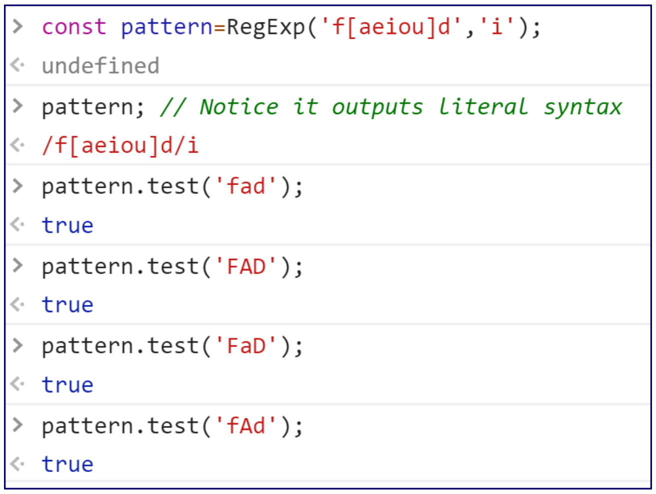
```
> const pattern=/fo\.d/;
<· undefined
> pattern.test('fo.d');
<· true
> pattern.test('food');
<· false
> pattern.test('fo4d');
<· false
```

❖ 4.2.9. Case-Insensitive Matches

The `i` flag makes a whole pattern match case insensitive.

- **Literal Syntax:** Flags are added after the end slash. For example, `/aeiou/i` matches all lowercase and uppercase vowels.
- **`RegExp()` Constructor Syntax:** Flags are added as the second argument. For example, `RegExp('abcde','i')` matches all lowercase and uppercase vowels.

The pattern `/f[aeiou]d/i` or `RegExp('f[aeiou]d', 'i')` can be found in "fad", "FAD", "FaD", and "fAd".

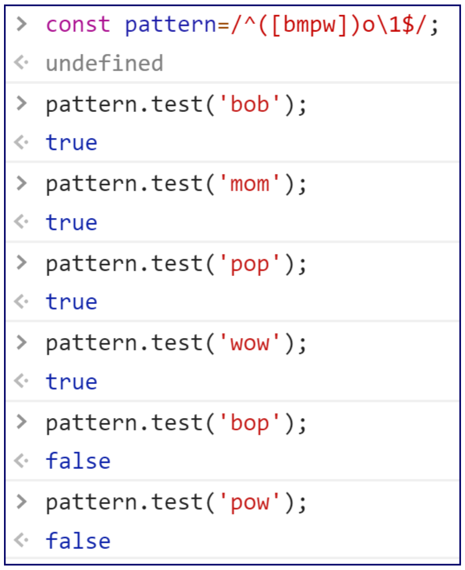


```
> const pattern=RegExp('f[aeiou]d','i');
<· undefined
> pattern; // Notice it outputs literal syntax
<· /f[aeiou]d/i
> pattern.test('fad');
<· true
> pattern.test('FAD');
<· true
> pattern.test('FaD');
<· true
> pattern.test('fAd');
<· true
```

4.3. Backreferences

Backreferences are special wildcards that refer back to a subpattern within a pattern. They can be used to make sure that two subpatterns match. The first subpattern in a pattern is referenced as `\1`, the second is referenced as `\2`, and so on.

For example, the pattern `/^([bmpw])o\1$/` matches "bob", "mom", "pop", and "wow", but not "bop" or "pow".



```
> const pattern=/^([bmpw])o\1$/;
<· undefined
> pattern.test('bob');
<· true
> pattern.test('mom');
<· true
> pattern.test('pop');
<· true
> pattern.test('wow');
<· true
> pattern.test('bop');
<· false
> pattern.test('pow');
<· false
```

A more practical example has to do with matching the delimiter in social security numbers, which are 9 digits long and separated into three parts: three digits, then two digits, then four digits. Examine the following regular expression:

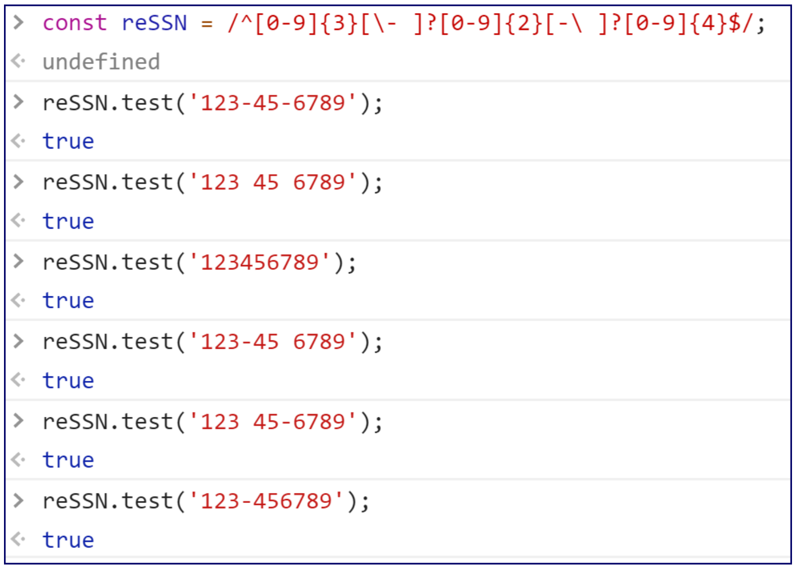
```
/^\d{3}([\- ]?)\d{2}([\- ]?)\d{4}$/
```

Within the caret (`^`) and dollar sign (`$`), which are used to specify the beginning and end of the pattern, there are three sequences of digits, optionally separated by a hyphen or a space. Note that the hyphen needs to be escaped with a backslash because dashes are special characters in regular expressions. This pattern will be matched in all of the following strings (and more):

1. 123-45-6789
2. 123 45 6789
3. 123456789
4. 123-45 6789

5. 123 45-6789
6. 123-456789

The last three strings are not ideal, but they do match the pattern.

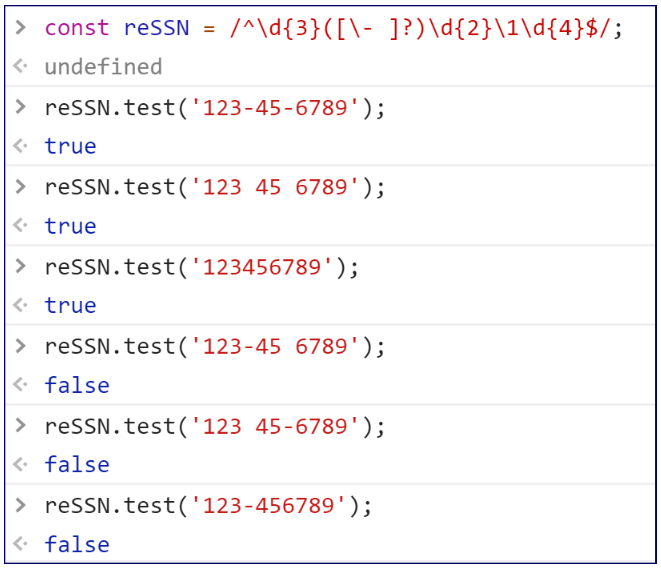


```
> const reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[-\ ]?[0-9]{4}$/;
< undefined
> reSSN.test('123-45-6789');
< true
> reSSN.test('123 45 6789');
< true
> reSSN.test('123456789');
< true
> reSSN.test('123-45 6789');
< true
> reSSN.test('123 45-6789');
< true
> reSSN.test('123-456789');
< true
```

Backreferences can be used to make sure that the second delimiter matches the first delimiter. The regular expression would look like this:

```
/^\d{3}([\- ]?)\d{2}\1\d{4}$/
```

The `\1` refers back to the first subpattern. Only the first three strings listed above match this regular expression.



```
> const reSSN = /^\d{3}([\- ]?)\d{2}\1\d{4}$/;
<· undefined
> reSSN.test('123-45-6789');
<· true
> reSSN.test('123 45 6789');
<· true
> reSSN.test('123456789');
<· true
> reSSN.test('123-45 6789');
<· false
> reSSN.test('123 45-6789');
<· false
> reSSN.test('123-456789');
<· false
```

4.4. Form Validation with Regular Expressions

Regular expressions are often used to create stronger form validation. This can often be done using the `pattern` attribute in form fields. Consider the HTML `email` type. While it does force the user to enter a syntactically valid email address, it is not very restrictive. For example, the following would be considered valid "a@a" and would pass as a valid email address. You may wish to have your validation be a little stricter than that. For example, you might require the following sequence. The regular expression matching each sequence part is shown in parentheses:

1. One or more of any of the following characters: letters, numbers, underscores, dashes, and periods.
   - Matching regular expression: `([\w\-\.])+`
2. An @ sign.
   - Matching regular expression: `@`
3. One or more of any of the following characters: letters, numbers, underscores, dashes, and periods.
   - Matching regular expression: `([\w\-\.])+`

4. A period followed by two to four letters.
    - Matching regular expression: `\.([A-Za-z]{2,4})`

The full regular expression looks like this:

```
([\w\-\.])+@([\w\-\.])+\.([A-Za-z]{2,4})
```

Email Regular Expression

Note that the regular expression shown above prevents some syntactically valid email addresses and allows for some invalid ones. Creating a robust pattern for an email address is quite complex. The site `https://emailregex.com` offers the following regular expression, which it claims works 99.99% of the time:

```
/^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-
9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/
```

As mentioned earlier, you can use regular expressions to validate email addresses, zip codes, usernames, passwords, or any number of other fields simply by using the `pattern` attribute in your HTML input fields, like this:

```
<input type="email"
  pattern="([\w\-\.])+@([\w\-\.])+\.([A-Za-z]{2,4})">
```

4.5. Cleaning Up Form Entries

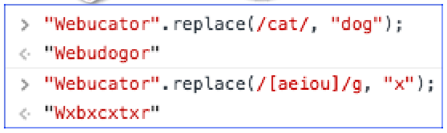
Regular expressions can also be used to clean up user entries immediately after they are entered. This can be done using a combination of regular expressions and the `replace()` method of a string object.

The replace() Method

The `replace()` method of a string takes two arguments: a regular expression and a replacement string. It replaces the first regular expression match in the string with the replacement string. If the `g` flag (for global) is used in the regular expression, it replaces all matches with the string.

```
"Webucator".replace(/cat/, "dog"); //returns Webudogor
"Webucator".replace(/[aeiou]/g, "x"); //returns Wxbxcxtxr
```

And here it is in the Google Chrome console:



```
> "Webucator".replace(/cat/, "dog");
< "Webudogor"
> "Webucator".replace(/[aeiou]/g, "x");
< "Wxbxcxtxr"
```

With the `replace()` method, it is also possible to replace a matched pattern with a new string made up of submatches from the pattern. The following example illustrates this.

Demo 4.1: RegularExpressions/Demos/ssn-cleaner.html

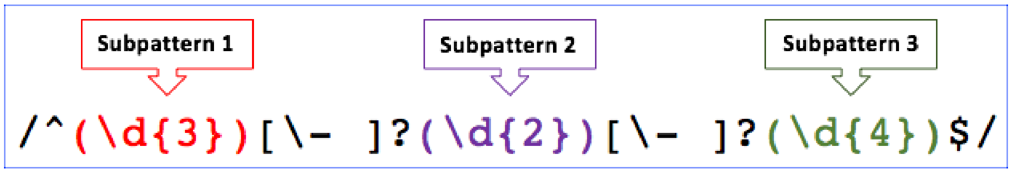
```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="../normalize.css">
<link rel="stylesheet" href="../styles.css">
<title>SSN Cleaner</title>
<script>
function cleanSSN(ssn) {
  const reSSN = /^(\d{3})[\- ]?(\d{2})[\- ]?(\d{4})$/;
  if (reSSN.test(ssn)) {
    const cleanedSsn = ssn.replace(reSSN, "$1-$2-$3");
    return cleanedSsn;
  } else {
    alert("INVALID SSN");
    return ssn;
  }
}

window.addEventListener('load', function(e) {
  const btn = document.getElementById('clean');
  const ssn = document.getElementById('ssn');
  btn.addEventListener('click', function(e) {
    ssn.value = cleanSSN(ssn.value);
    e.preventDefault(); // prevent form submission
  });
});
</script>
</head>
<body>
  <form>
    <input id="ssn" name="ssn" size="20">
    <button id="clean">Clean SSN</button>
  </form>
</body>
</html>
```

Code Explanation

The `cleanSSN()` function is used to "clean up" a social security number. The regular expression contained in `reSSN` contains three subexpressions, denoted with parentheses:

```
^(\d{3})[\- ]?(\d{2})[\- ]?(\d{4})$
```

1. `(\d{3})`
2. `(\d{2})`
3. `(\d{4})`



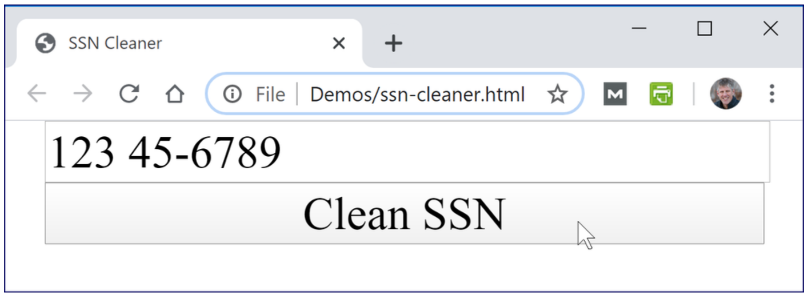
Within the second argument of the `replace()` method, these subexpressions can be referenced as `$1`, `$2`, and `$3`, respectively.

When the user clicks the "Clean SSN" button, the `cleanSSN()` function is called. This function first uses a regular expression to test that the user-entered value is a valid social security number. If it is, it then cleans it up with the following line of code, which dash-delimits the three substrings matching the subexpressions.

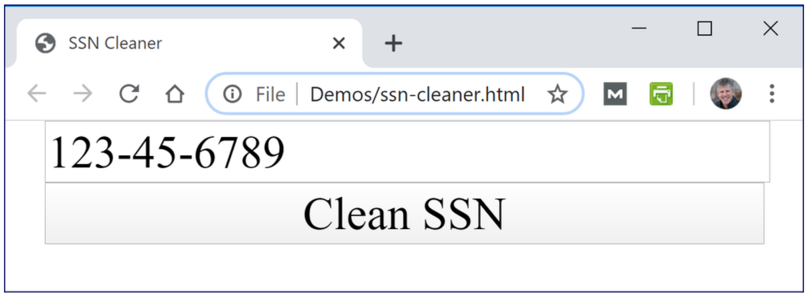
```
const cleanedSsn = ssn.replace(reSSN, "$1-$2-$3");
```

It then returns the cleaned-up social security number.

For example, in the following form, the social security number is entered with one space and one dash:



After clicking the **Clean SSN** button, the social security number gets formatted with two dashes:


SSN Cleaner
File | Demos/ssn-cleaner.html
123-45-6789
Clean SSN

Exercise 7: Cleaning Up Form Entries

15 to 25 minutes

In this exercise, you will create a function for validating and formatting a phone number. The phone number should have the following format:

1. An optional open parentheses.
2. Three digits.
3. An optional close parentheses.
4. An optional dash, dot or space.
5. Three digits.
6. An optional dash, dot or space.
7. Four digits.

Here are some examples of valid phone numbers:

1. `5551234567`
2. `(555) 123 4567`
3. `555-123-4567`
4. `555 123 4567`

You are to do the following:

1. Open `RegularExpressions/Exercises/phone-cleaner.html` for editing.
2. Add a `pattern` attribute to the `phone` field with a regular expression matching the pattern described above.
3. Write a `cleanPhone()` function that takes one argument: the field containing a phone number. It should check to see that the field contains a valid phone number.
   A. If it does, it should return a cleaned-up version of the phone number in the format: `(555) 123-4567`.
   B. If it does not, it should alert "INVALID PHONE" and return the current unchanged invalid phone value.
4. Test your solution in a browser.

Solution: RegularExpressions/Solutions/phone-cleaner.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="../normalize.css">
<link rel="stylesheet" href="../styles.css">
<title>Phone Cleaner</title>
<script>
function cleanPhone(phone) {
  const rePhone = RegExp(phone.pattern);
  const phoneNum = phone.value;
  const cleanedPhone = phoneNum.replace(rePhone, "($1) $2-$3");
  if (rePhone.test(phoneNum)) {
    return cleanedPhone;
  } else {
    alert("INVALID PHONE");
    return phoneNum;
  }
}

window.addEventListener('load', function(e) {
  const btn = document.getElementById('clean');
  const phone = document.getElementById('phone');
  btn.addEventListener('click', function(e) {
    phone.value = cleanPhone(phone);
    e.preventDefault(); // prevent form submission
  });
});
</script>
</head>
<body>
  <form>
    <input type="tel" id="phone" name="phone" size="20"
    pattern="^\(?(\d{3})\)?[\-\. ]?(\d{3})[\-\. ]?(\d{4})$">
    <button id="clean">Clean Phone</button>
  </form>
</body>
</html>
```

4.6. A Slightly More Complex Example

Some phone numbers are given as a combination of numbers and letters (e.g, 877-WEBUCATE). As is the case with 877-WEBUCATE, such numbers often have an extra character just to make the word complete.

In the following demo, we will:

1. Add a function called `convertPhone()` that:
    - Strips all characters that are not numbers or letters.
    - Converts all letters to numbers using the following rules:
        - ABC -> 2
        - DEF -> 3
        - GHI -> 4
        - JKL -> 5
        - MNO -> 6
        - PQRS -> 7
        - TUV -> 8
        - WXYZ -> 9

      Notice that the regular expressions used in the conversions include both the `g` flag (for **g**lobal) so that all matches get replaced, and the `i` flag, so that the search is case-insensitive.
    - Passes the first 10 characters of the resulting string, and the pattern to use to the `cleanPhone()` function.
    - Returns the resulting string.
2. Modify the form, so that it calls `convertPhone()` rather than `cleanPhone()`.

Demo 4.2: RegularExpressions/Demos/phone-converter.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="../normalize.css">
<link rel="stylesheet" href="../styles.css">
<title>Phone Converter</title>
<script>
function cleanPhone(phone, pattern) {
  const cleanedPhone = phone.replace(pattern, "($1) $2-$3");
  return cleanedPhone;
}

function convertPhone(phone) {
  const rePhone = /^\(?(\d{3})\)?[\-\. ]?([A-Za-z\d]{3})[\-\. ]?([A-Za-z\d]{4})/;
  if (!rePhone.test(phone.value)) {
    alert("INVALID PHONE");
    return phone.value;
  }
  let convertedPhone = phone.value.replace(/[^A-Za-z\d]/g, "");
  convertedPhone = convertedPhone.replace(/[ABC]/gi, "2");
  convertedPhone = convertedPhone.replace(/[DEF]/gi, "3");
  convertedPhone = convertedPhone.replace(/[GHI]/gi, "4");
  convertedPhone = convertedPhone.replace(/[JKL]/gi, "5");
  convertedPhone = convertedPhone.replace(/[MNO]/gi, "6");
  convertedPhone = convertedPhone.replace(/[PQRS]/gi, "7");
  convertedPhone = convertedPhone.replace(/[TUV]/gi, "8");
  convertedPhone = convertedPhone.replace(/[WXYZ]/gi, "9");
  phone.value = convertedPhone.substring(0, 10);

  return cleanPhone(phone.value, rePhone);
}

window.addEventListener('load', function(e) {
  const btn = document.getElementById('clean');
  const phone = document.getElementById('phone');
  btn.addEventListener('click', function(e) {
    phone.value = convertPhone(phone);
    e.preventDefault(); // prevent form submission
  });
});
</script>
</head>
```

```
<body>
  <form novalidate>
    <input type="tel" id="phone" name="phone" size="20">
    <button id="clean">Clean Phone</button>
  </form>
</body>
</html>
```

Conclusion

In this lesson, you have learned to work with regular expressions to validate and to clean up form entries.

LESSON 5

Node.js and Server-side Form Validation

Topics Covered

☑ Installing and running the Node.js.

☑ Server side-form validation.

Introduction

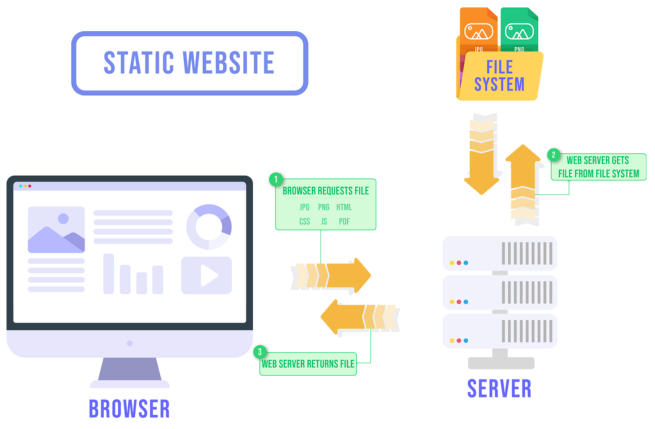
In this lesson, there is quite a lot to understand before we give you any exercises. However, you should follow along with all the demos, reviewing every one in your editor and running them all in the browser. If you do this, you should have a good grasp of server-side programming and Node.js before you get to the form validation exercises.

5.1. Welcome to the Server-side

Up to this point we have only used client-side JavaScript. Client-side JavaScript is executed by the JavaScript engine of the browser, but browsers aren't the only software that can include JavaScript engines. Node.js, pronounced to rhyme with "Toads say yes" and often just referred to as "Node," uses a JavaScript engine to create web servers.

❖ 5.1.1. What is a web server?

The first step to understanding server-side programming is to understand how web server software works. The diagram below shows how a web server delivers static pages, such as HTML, JavaScript, CSS, image files, audio and video files, PDFs, all of which browsers have a built-in way of handling; and other files that can be downloaded but not handled by the browser, such as Microsoft Word documents, zip files, and executables. All these files, both the ones the browser handles and the ones it just downloads, are *static*, because they are simply fetched by the web server and returned to the client without any processing on the server side.

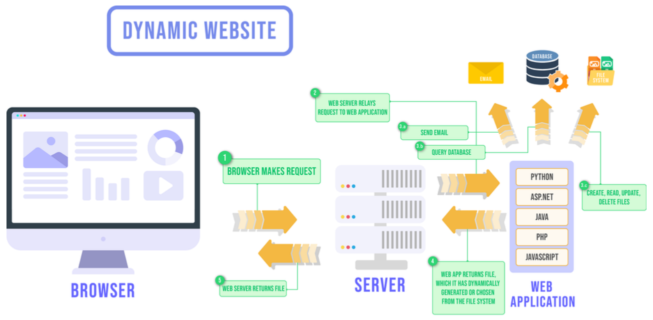


❖ 5.1.2. Dynamic Websites

Dynamic websites are websites that do more than just fetch and return files. They have software on the server that reviews the client request before deciding what to do. Depending on the client request, the server may just return a static file or it may perform any number of processes on the server before returning a dynamically created file to the client. Here are some examples of what a dynamic site might do after receiving a request and before returning dynamically created content to the client:

1. Perform a database search and return a list of search results.
2. Log a user into a website by checking the database for the user's credentials.
3. Redirect the user to a login page if the user requests a members-only page.
4. Record a user's support request in a database, email the user a friendly "we-will-be-in-touch-soon" message and the auto-generated support ticket number, email the support team letting them know a new request has come in, and return a dynamically created HTML page with a friendly "we-will-be-in-touch-soon" message and the auto-generated support ticket number.

Web servers can have access to all sorts of software on the computer on which they sit and can even reach across networks to make requests of other servers, so the variety of tasks they can perform is infinite. The diagram below shows how a dynamic website works:


DYNAMIC WEBSITE
BROWSER
1 BROWSER MAKES REQUEST
2 WEB SERVER RELAYS REQUEST TO WEB APPLICATION
3.a SEND EMAIL
3.b QUERY DATABASE
3.c CREATE, READ, UPDATE, DELETE FILES
EMAIL
DATABASE
FILE SYSTEM
4 WEB APP RETURNS FILE, WHICH IT HAS DYNAMICALLY GENERATED OR CHOSEN FROM THE FILE SYSTEM
5 WEB SERVER RETURNS FILE
SERVER
PYTHON
ASP.NET
JAVA
PHP
JAVASCRIPT
WEB APPLICATION

At the time of this writing, the most widely used web servers[8] are:

1. Apache
2. nginx
3. Microsoft-IIS

And the most widely used server-side programming languages[9] are:

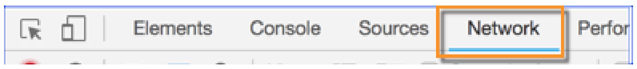
1. PHP
2. ASP.NET
3. Java
4. Ruby
5. Python
6. Scala
7. JavaScript
8. ColdFusion
9. Perl

8. https://w3techs.com/technologies/overview/web_server
9. https://w3techs.com/technologies/overview/programming_language

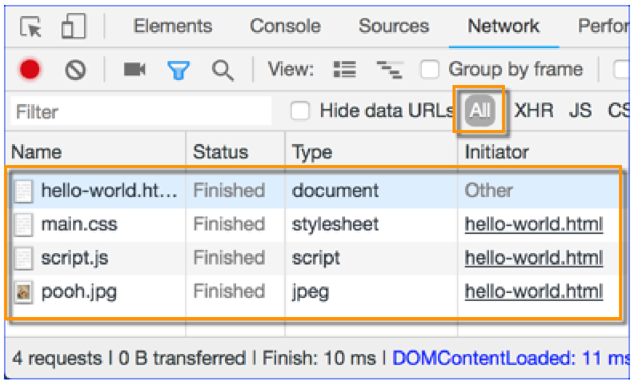
5.2. Google Chrome DevTools: Network Tab

Google Chrome DevTools Network tab shows which files are delivered when the browser makes a request. Let's first take a look at what it shows when we request a file from our local file system without going through a server:

1. Open `Node/Demos/no-server/hello-world.html` from your class files in Google Chrome by double-clicking the file or right-clicking and selecting the option to open with Google Chrome. This will give you a local view of the file, meaning the file will not be delivered through a web server. The URL in the browser's location bar should not begin with `http`. If it does, close the file and re-open it by navigating to the folder in your file system (not in Visual Studio Code) and double-clicking it.
2. Open Chrome DevTools and select the **Network** tab:



3. Now reload the page and look at the **Network** tab:

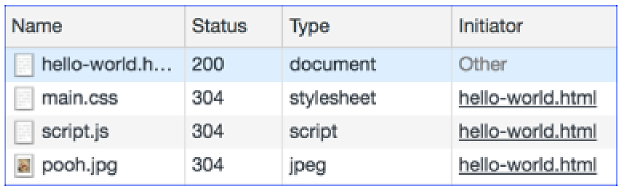


Name	Status	Type	Initiator
hello-world.ht...	Finished	document	Other
main.css	Finished	stylesheet	hello-world.html
script.js	Finished	script	hello-world.html
pooh.jpg	Finished	jpeg	hello-world.html

4 requests | 0 B transferred | Finish: 10 ms | DOMContentLoaded: 11 ms

This shows what documents were delivered to the browser, their status, what type of documents they were, and what initiated the delivery.

4. Here are the same static files delivered through a web server rather than opened locally. We will look further at how this was delivered soon:



Name	Status	Type	Initiator
hello-world.h...	200	document	Other
main.css	304	stylesheet	hello-world.html
script.js	304	script	hello-world.html
pooh.jpg	304	jpeg	hello-world.html

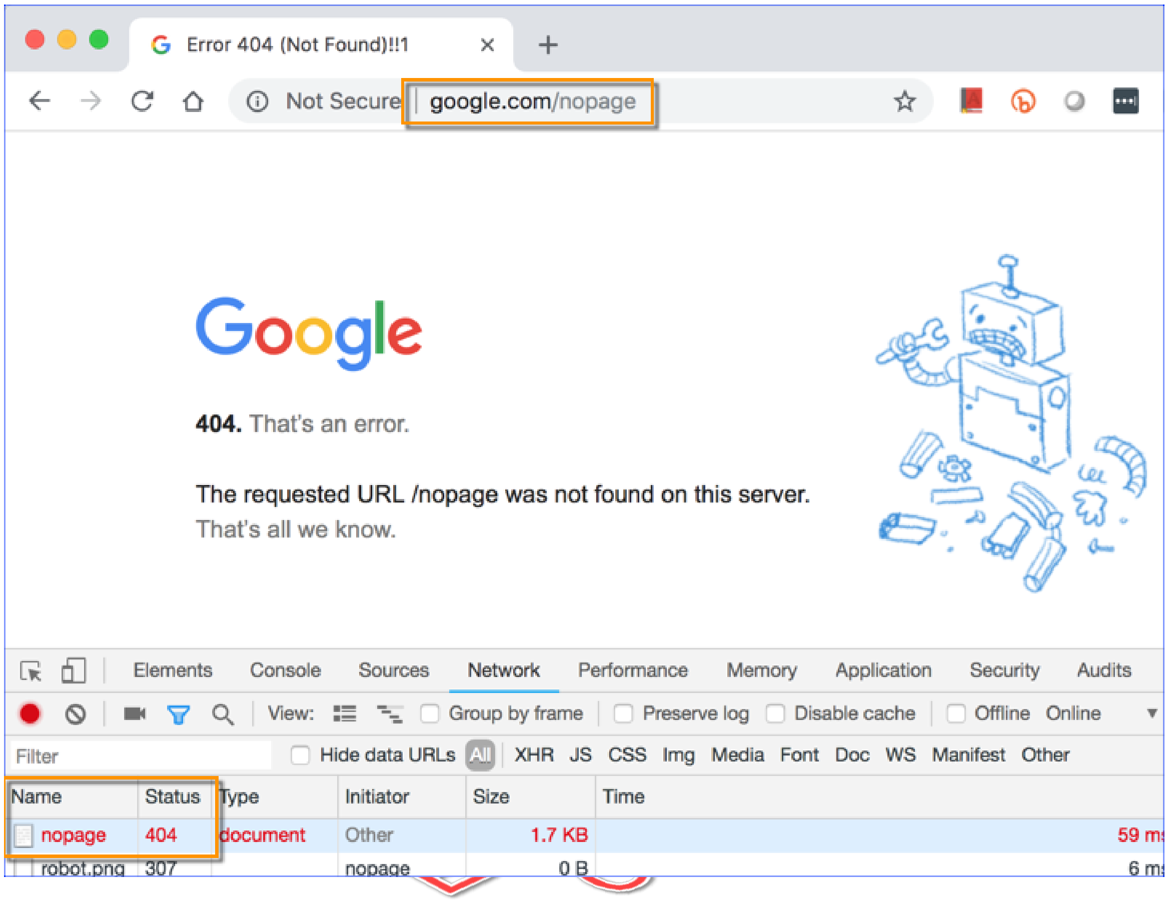
The only difference is the **Status** column. The web server sends back a status code to let the client know the status of the file. The 200 status code means that everything is fine. The 304 status code means that the file hasn't been modified since the last time it was requested, so the browser can use the version it has in cache if it has one.

❖ 5.2.1. Status Codes

The most common status codes returned by a web server are listed below along with their meanings:

- 200 **OK**.
- 301 **Moved Permanently**. The file used to be at this URL, but it isn't anymore.
- 304 **Not Modified**. The file hasn't changed from the last time it was sent to the client.
- 400 **Bad Request**. Something about the way the request was made has baffled the web server.
- 401 **Unauthorized**. You have to be logged in to access this file.
- 403 **Forbidden**. You can't have this even if you are logged in.
- 404 **Not Found**. There is no file here.
- 500 **Internal Server Error**. Something went wrong on the server.
- 503 **Service Unavailable**. The web server is down or overloaded.

As the server-side developer, you have the ability to return these status codes and to decide what pages get returned with them. For example, it is common to return a special "404 Page Not Found" page when the user navigates to a URL on your website that doesn't exist. The following screenshot shows how Google handles this:



Error 404 (Not Found)!!1

Not Secure | google.com/nopage

Google

404. That's an error.

The requested URL /nopage was not found on this server. That's all we know.

Name	Status	Type	Initiator	Size	Time
nopage	404	document	Other	1.7 KB	59 ms
robot.png	307		nopage	0 B	6 ms

Notice the 404 status on the **Network** tab.

Now let's take a look at how we can easily create and start a web server with JavaScript using Node.js.

5.3. Welcome to Node.js

Node.js is a JavaScript runtime that makes it simple to create a powerful web server.

❖ 5.3.1. Installing Node.js

Installing Node.js is easy, but the process is different for different operating systems. Please refer to the following web pages for installing and running Node.js:

- For Mac: `https://www.webucator.com/article/how-to-install-nodejs-on-a-mac/`
- For Windows: `https://www.webucator.com/article/how-to-install-nodejs-on-windows/`

❖ 5.3.2. package.json

The Node.js installer also installs **npm**, which is a JavaScript package manager. Among other things, npm is used to create, install, update, and run JavaScript packages. A package is simply a directory with files in it, including one file named `package.json`, which contains metadata about the project. Below we show the contents of a `package.json` file for the most basic Node.js application:

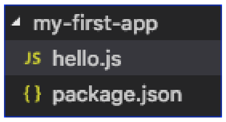
```
{
   "name": "myfirstapp",
   "version": "1.0.0"
}
```

This is an object literal written in JSON, which we will learn about later in the course. While the `package.json` file can specify many properties, the only essential properties are `name` and `version`.

❖ 5.3.3. Our First App

Let's go ahead and create our first Node.js application.

1. In your editor, navigate to the `Node/Demos` folder.
2. Within that folder, create a new folder called `my-first-app` and within that create two files:
   A. `package.json`
   B. `hello.js`
3. Your directory should now look like this:



```
my-first-app
  hello.js
  package.json
```

4. Open `package.json`, enter the following code and save:

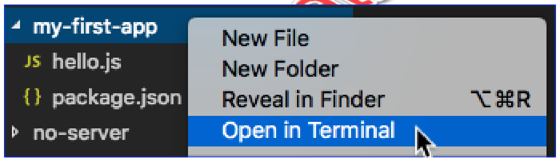
```
{
  "name": "my-first-app",
  "version": "1.0.0",
  "scripts": {
    "start": "node hello.js"
  }
}
```

5. Open `hello.js`, enter the following code and save:

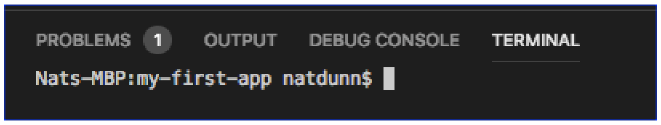
```
console.log('Hello, world!');
```

6. Now open the **Windows Command Prompt** or the **Mac Terminal**. From here on out, we will refer to both as the *prompt*.

   - In Visual Studio Code, you can open the prompt by right-clicking the directory in **Explorer** and selecting **Open in Integrated Terminal**.

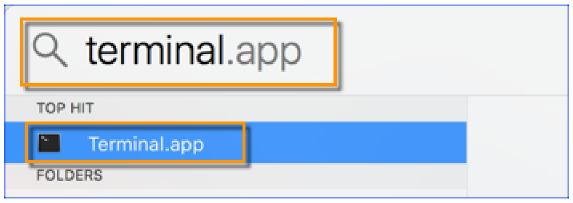


     This will open the prompt at the directory.

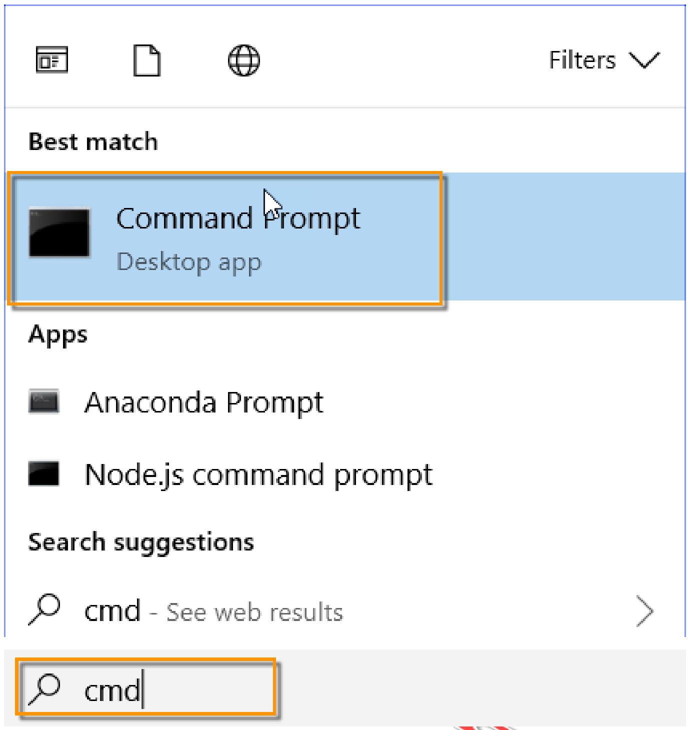


```
Nats-MBP:my-first-app natdunn$
```

   - Or you can access the prompt directly. On a Mac, press **Cmd+Space**, type "Terminal", and press **Enter**:



     On Windows, type "cmd" in the **Windows Search Bar** and select **Command Prompt**:



You will then need to navigate to the `Node/Demos/my-first-app` directory using the `cd` (for **c**hange **d**irectory) command. The exact path will depend on where you installed the class files. It will look something like this:

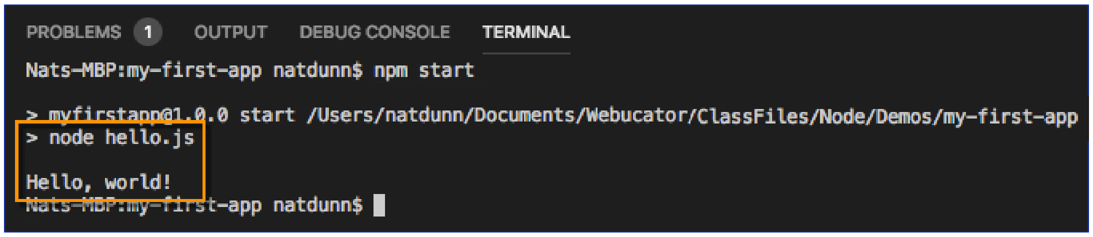
- **Mac**:

```
cd /Users/natdunn/Documents/Webucator/ClassFiles/Node/Demos/my-
first-app
```

- **Windows**:

```
cd \Webucator\ClassFiles\Node\Demos\my-first-app
```

7. Once at the proper directory, type `npm start` at the prompt and press **Enter**:



```
Nats-MBP:my-first-app natdunn$ npm start

> myfirstapp@1.0.0 start /Users/natdunn/Documents/Webucator/ClassFiles/Node/Demos/my-first-app
> node hello.js

Hello, world!
Nats-MBP:my-first-app natdunn$
```

Notice that this runs `node hello.js`, which simply logs "Hello, world!" to the console. Note that the JavaScript code in `hello.js` is not running within the browser. It is running in the Node.js's JavaScript engine. As such, the console is not the browser's console, but rather the prompt itself.

❖ 5.3.4. What does npm start do?

The `npm start` command looks at the `package.json` file for a "start" script. That's this piece of the `package.json` code:

```
"scripts": {
  "start": "node hello.js"
}
```

If it does not find a "start" script, it looks in the directory for a `server.js` file. If it finds it, it runs `node server.js`. If it does not find it, it will error.

5.4. Our First Web App

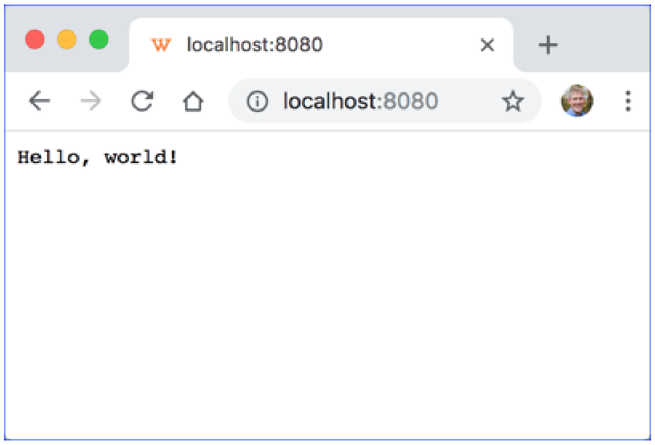
Now it's time to build our first Node.js web application, which will just send a "Hello, world!" message to the browser in plain text.

1. Navigate to `Node/Demos/minimum` at the prompt.
2. Type `npm start` and press **Enter**:

```
Nats-MBP:minimum natdunn$ npm start

> minimum@1.0.0 start /Users/natdunn/Documents/Webucator/ClassFiles/Node/Demos/min  ↵
imum
> node server.js
```

3. Notice that this runs `node server.js`. The server is now running, waiting to serve up content as we view it in the browser.

4. Now go to `http://localhost:8080` in Google Chrome. You should see a page like this one:



```
Hello, world!
```

The `Node/Demos/minimum` contains only three files: `package.json` and `server.js` and a commented version of `server.js`, `server-commented.js`, which isn't used in the application. The first two are shown below:

Demo 5.1: Node/Demos/minimum/package.json

```
{
  "name": "minimum",
  "description": "A very basic web application",
  "version": "1.0.0"
}
```

Code Explanation

Notice that this file has no "start" script. As such, it looks in the directory for a `server.js` file and runs that.

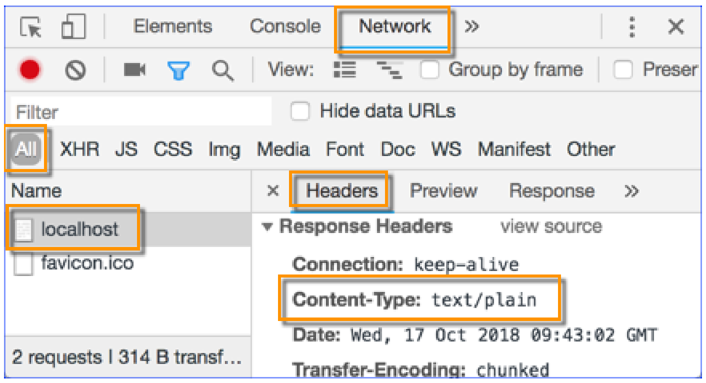
Demo 5.2: Node/Demos/minimum/server.js

```
const http = require('http');
const server = http.createServer( function(request, response) {
    const head = {
        'Content-Type': 'text/plain'
    }
    response.writeHead(200, head);
    response.end('Hello, world!');
});

server.listen(8080);
```

Code Explanation

This is where the magic happens. The `server.js` file:

1. Requires the `http` module, which
2. Creates a server and sends a very simple response. Before looking at the code, let's look at how Google Chrome receives that response:
    A. Back in Google Chrome, open the Chrome DevTools **Network** tab and refresh the page.
    B. Make sure **All** is selected, meaning that it will list all files sent from the server.
    C. Click **localhost** in the bottom-left area.
    D. Make sure the **Headers** tab is selected in the bottom-right area. This tab shows both the request headers (headers sent from the client to the server when making the request) and the response headers (headers sent from the server to the client when responding to the request).
    E. Scroll down to the **Response Headers** section.



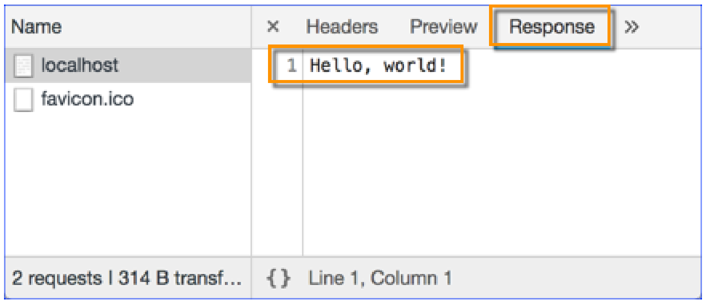
▾ Response Headers

Connection: keep-alive
Content-Type: text/plain
Date: Wed, 17 Oct 2018 09:43:02 GMT
Transfer-Encoding: chunked

Notice that the `Content-Type` is “text/plain”. In our `server.js` file, you will see the following code:

```
const head = {
  'Content-Type': 'text/plain'
}
```

F. Now click the **Response** tab:



```
Hello, world!
```

Notice that it just returns the text “Hello, world!” without any HTML tags. That’s because of this line in the `server.js` file:

```
response.end('Hello, world!');
```

Now let’s look at the commented version of `server.js`:

Demo 5.3: Node/Demos/minimum/server-commented.js

```
1.   // Require the http module
2.   const http = require('http');
3.
4.   // Create the HTTP server
5.   const server = http.createServer( function(request, response) {
6.       const head = {
7.           'Content-Type': 'text/plain'
8.       }
9.       // Send the response header to the request
10.      response.writeHead(200, head);
11.      // Send the response body and indicate that message is complete
12.      response.end('Hello, world!');
13.  });
14.
15.  // Start the HTTP server listening for connections
16.  server.listen(8080);
```

Code Explanation

Things to notice:

1. On line 2, we require the `http` module and assign it to the `http` constant.
2. Starting on line 5, we use the `createServer()` method of `http` to create the web server and pass it a callback function, which will run every time a request is made.
3. On line 10, we use `response.writeHead()` to send the response header to the request. The first argument (`200`) is the status code and the second argument (`head`) is the headers being sent to the browser.
4. On line 12, we call `response.end('Hello, world!')`, which does two things:
   A. Sends the response body to the request.
   B. Lets the server know the response is complete.
5. On line 16, we tell the server to start listening for requests on port 8080.

Choosing the Port

If you get an error saying that port 8080 is already in use (e.g., `Error: listen EADDRINUSE :::8080`), try a different port, such as 3000 or 8000.

❖ 5.4.1. Stopping the Server

To stop the server, press **Ctrl+C** in the prompt.

5.5. Fat-arrow Functions

In JavaScript, it is common to use an abbreviated form of anonymous function when defining callback functions. Let's take a look at the event listening function we saw in `server.js`:

```
function(request, response) {
  const head = {
    'Content-Type': 'text/plain'
  }
  response.writeHead(200, head);
  response.end('Hello, world!');
}
```

This function can also be written like this:

```
(request, response) => {
    const head = {
    'Content-Type': 'text/plain'
  }
  response.writeHead(200, head);
  response.end('Hello, world!');
}
```

The only difference is that the word `function` is removed and a "fat arrow" is inserted after the parameters.

Here is the `server.js` file using a fat-arrow function:

Demo 5.4: Node/Demos/minimum-fat-arrow/server.js

```
const http = require('http');
const server = http.createServer((request, response) => {
    const head = {
        'Content-Type': 'text/plain'
    }
    response.writeHead(200, head);
    response.end('Hello, world!');
});

server.listen(8080);
```

We will use fat-arrow functions for callback functions for the rest of this lesson.

5.6. Sending a Response with HTML

In the last demo, we showed how to send plain text in the response to the browser. Sending HTML is very similar; we just need to change the `Content-Type`:

Demo 5.5: Node/Demos/minimum-html/server.js

```
const http = require('http');
const server = http.createServer((request, response) => {
    const head = {
        'Content-Type': 'text/html; charset=UTF-8'
    }
    const html = `<!DOCTYPE html>
    <html lang="en">
    <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Hello, world!</title>
    </head>
    <body>
       <h1>Hello, world!</h1>
    </body>
    </html>`;

    response.writeHead(200, head);
    response.end(html);
    console.log(request.url);
});

server.listen(8080);
```

Code Explanation

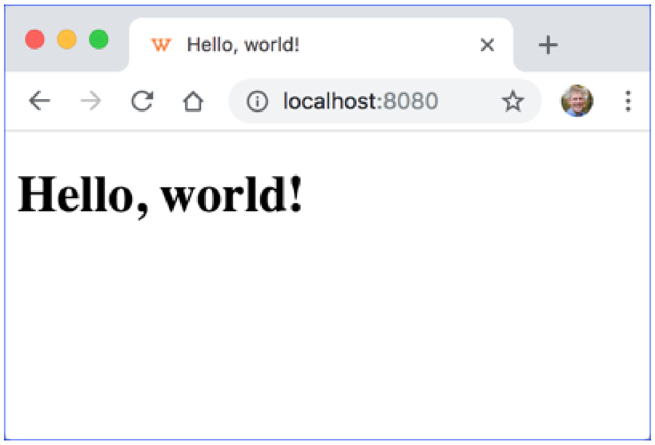
Things to notice:

1. We set the `'Content-Type'` key to `'text/html; charset=UTF-8'`.
2. We create an `html` variable to hold the HTML content that we will write to the body of the response. Note the use of back-ticks (`) to create a *template literal*. Template literals allow for multi-line strings and also can have embedded expressions using the following syntax: `${expression}`. We will see an example of this soon.
3. We send the `html` with the response using `response.end(html)`.
4. We log the requested URL.

To run this:

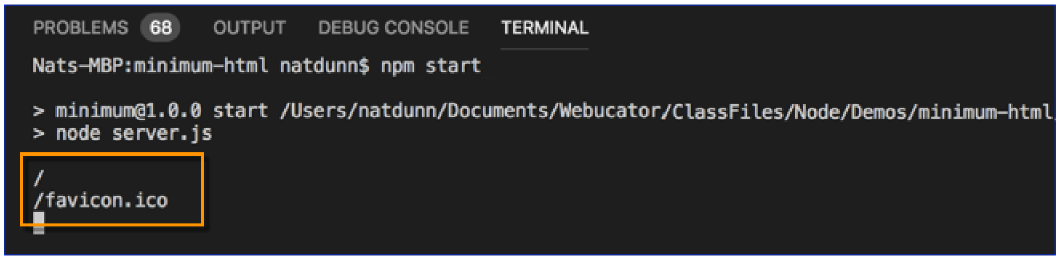
1. Navigate to `Node/Demos/minimum-html` at the prompt, type `npm start` and press **Enter**.

2. In your browser, navigate to `http://localhost:8080`. You should see a page that looks like this:



Hello, world!

3. Now return to the prompt to see what was logged to the console:

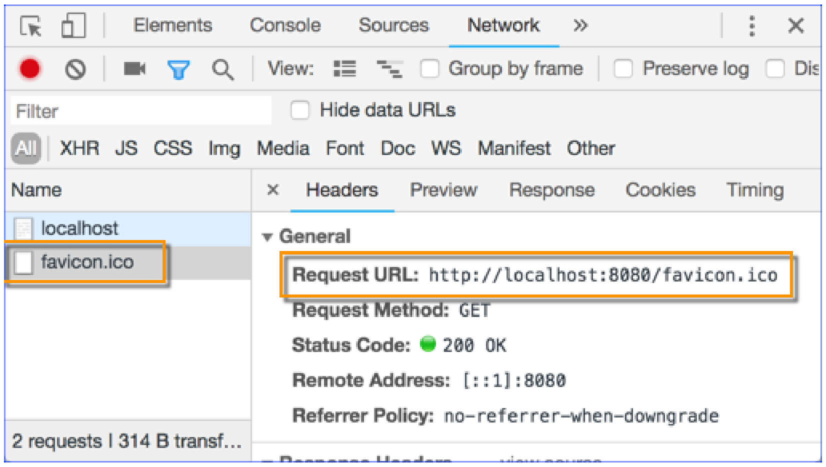


```
Nats-MBP:minimum-html natdunn$ npm start

> minimum@1.0.0 start /Users/natdunn/Documents/Webucator/ClassFiles/Node/Demos/minimum-html
> node server.js

/
/favicon.ico
```

From this log, you can see that, as a result of your browser visit, two requests were made to the server: one for the root page (`/`) and the other for `favicon.ico`. But we don't have anything in our code about any `favicon.ico` file. Where is that coming from? Stay tuned.

5.7. The favicon.ico Icon

As we saw from the console log in the last demo, the browser is making a request for `favicon.ico`. We can also see this request in the **Network** tab of Chrome DevTools:



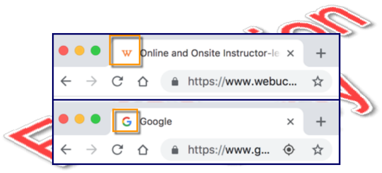
localhost
favicon.ico

2 requests | 314 B transf…

General

Request URL: http://localhost:8080/favicon.ico
Request Method: GET
Status Code: 200 OK
Remote Address: [::1]:8080
Referrer Policy: no-referrer-when-downgrade

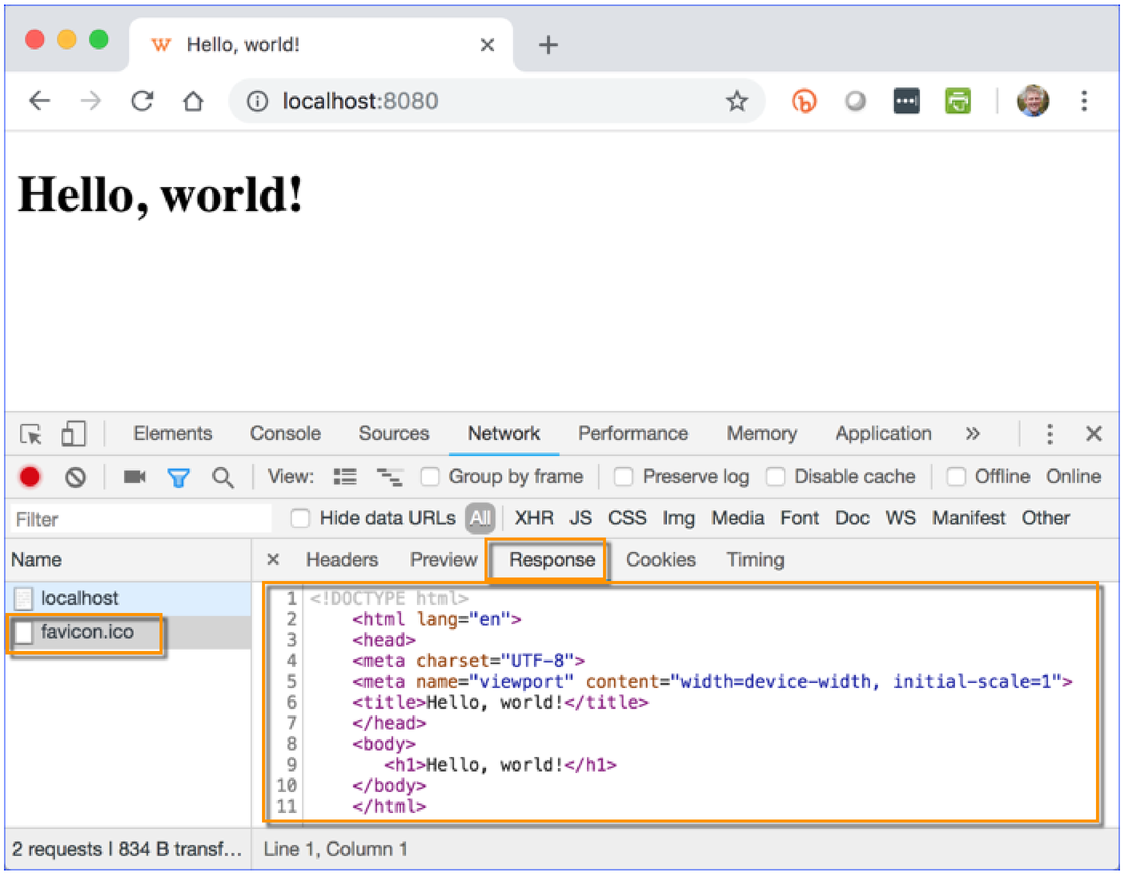
Among other things, the `favicon.ico` file is used by browsers, usually on the tab, as a logo-like identifier for the website:



While browsers can handle this differently, they generally make a request for `favicon.ico` when visiting a new site. This is why it shows up on the **Network** tab even when we didn't explicitly send it.

5.8. Simple Routing and 404 Pages

The way we have our `server.js` file written now, the same response will be sent regardless of the URL. You can see this by clicking `favicon.ico` and then clicking the **Response** tab:



Hello, world!

```
<!DOCTYPE html>
    <html lang="en">
    <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Hello, world!</title>
    </head>
    <body>
        <h1>Hello, world!</h1>
    </body>
    </html>
```

Try visiting `http://localhost:8080/foobar` or any URL beginning with `http://localhost:8080`. They will all return the Hello-world page.

Let's fix this with some simple routing:

Demo 5.6: Node/Demos/routing/server.js

```
const http = require('http');
const server = http.createServer((request, response) => {
    const head = {
        'Content-Type': 'text/html; charset=UTF-8'
    }

    let status = 200;
    let title, body;
    switch(request.url) {
        case '/':
           title = 'Welcome!';
           body = `<h1>Welcome to our Home Page!</h1>
           <a href="contact">Contact us</a>`;
           break;
        case '/contact':
            title = 'Contact Us';
            body = `<h1>Contact Us</h1>
            <p>Call us at 555-555-5555.</p>
            <a href="/">Home</a>`;
            break;
        default:
            status = 404;
            title = '404: Page Not Found';
            body = `<h1>404: Page Not Found</h1>
            <p>Sorry, that page doesn't exist.</p>
            <a href="/">Home</a>`;
    }

    const html = `<!DOCTYPE html>
    <html lang="en">
    <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>${title}</title>
    </head>
    <body>${body}</body>
    </html>`;

    response.writeHead(status, head);
    response.end(html);

    console.log(request.url);
});

```

```
45.   server.listen(8080);
```

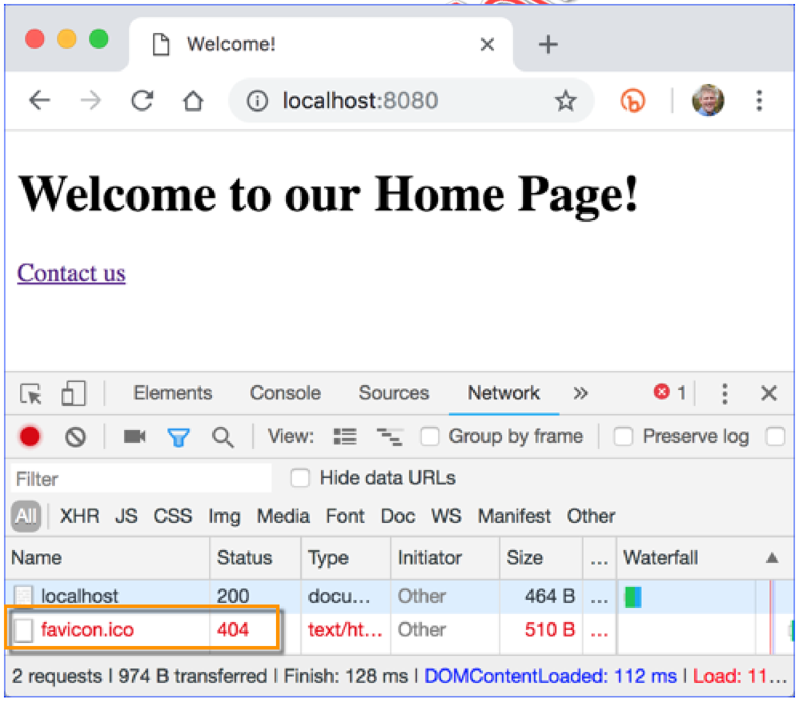
Code Explanation

Things to notice:

1. We use a `switch / case` statement to set variables for the `status` (set to 200 as a default) , `title`, and `body` based on the requested URL. Note that the switch statement defaults to a 404 page if we get an unexpected value.
2. The expressions `${title}` and `${body}` are used within the `html` variable template literal (denoted with back-ticks).

To run this:

1. Navigate to `Node/Demos/routing` at the prompt, type `npm start` and press **Enter**.
2. In your browser, navigate to `http://localhost:8080`. You should see a page that looks like this:



Welcome to our Home Page!

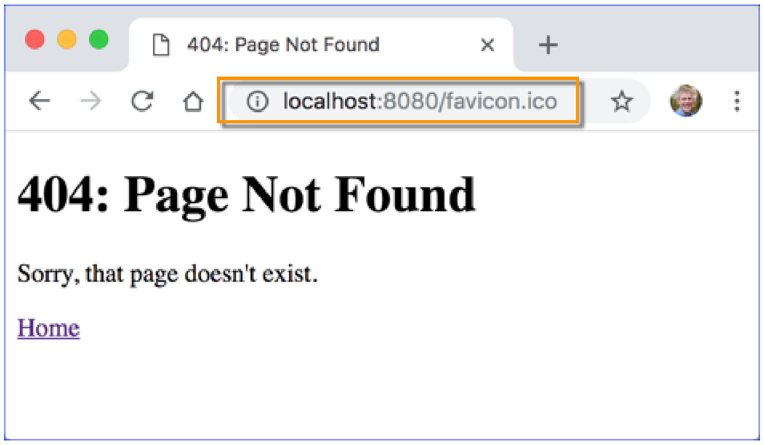
Contact us

Name	Status	Type	Initiator	Size	...	Waterfall
localhost	200	docu...	Other	464 B	...	
favicon.ico	404	text/ht...	Other	510 B	...	

2 requests | 974 B transferred | Finish: 128 ms | DOMContentLoaded: 112 ms | Load: 11...

Notice the **Network** tab. The request for `/favicon` now returns a 404 status.

3. Now explicitly point your browser to `http://localhost:8080/favicon`. You will see the 404 page you created in `server.js`:



404: Page Not Found

Sorry, that page doesn't exist.

Home

❖ 5.8.1. Delivering favicon.ico

Let's add one more route to our routing example to deliver the `favicon.ico` file.

Demo 5.7: Node/Demos/routing-favicon/server.js

```
const http = require('http');
const fs = require('fs');

const server = http.createServer((request, response) => {
    let head = {
        'Content-Type': 'text/html; charset=UTF-8'
    }

    let status = 200;
    let title, body;
    let img;
    switch(request.url) {
        case '/':
           title = 'Welcome!';
           body = `<h1>Welcome to our Home Page!</h1>
           <a href="contact">Contact us</a>`;
           break;
        case '/contact':
            title = 'Contact Us';
            body = `<h1>Contact Us</h1>
            <p>Call us at 555-555-5555.</p>
            <a href="/">Home</a>`;
            break;
        case '/favicon.ico':
            img = fs.readFileSync('./favicon.ico');
            head = {
                'Content-Type': 'image/x-icon'
            }
            break;
        default:
            status = 404;
            title = '404: Page Not Found';
            body = `<h1>404: Page Not Found</h1>
            <p>Sorry, that page doesn't exist.</p>
            <a href="/">Home</a>`;
    }

    response.writeHead(status, head);
    if (typeof img === 'undefined') {
        const html = `<!DOCTYPE html>
        <html lang="en">
        <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,initial-scale=1">
```

```
            <title>${title}</title>
            </head>
            <body>${body}</body>
            </html>`;
            response.end(html);
        } else {
            response.end(img, 'binary');
        }

        console.log(request.url);
    });

    server.listen(8080);
```

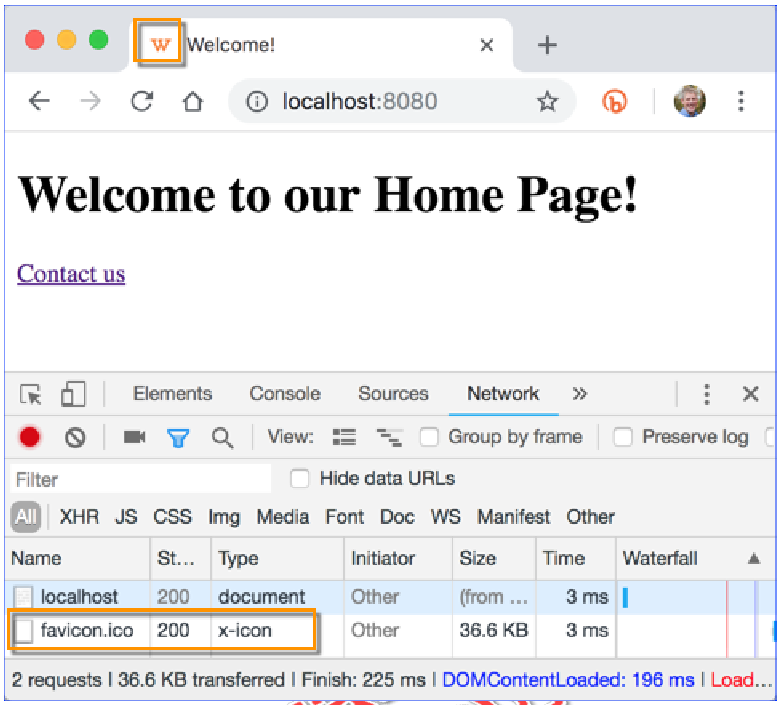
Code Explanation

Things to notice:

1. We require `fs`, which is Node.js's File System module.
2. We include a new case to check to see if the requested URL is `/favicon.ico`. If it is we read in the `favicon.ico` file, which is stored in the same directory, and we change the `Content-type` to `image/x-icon`.
3. Finally, we do a check to see if the `img` variable has been defined. If it has not, we respond with an HTML page. If it has we respond with our favicon.ico image.

To run this:

1. Navigate to `Node/Demos/routing-favicon` at the prompt, type `npm start` and press **Enter**.
2. In your browser, navigate to `http://localhost:8080/`. This should display the icon both on the tab and in the **Network** tab of Chrome DevTools:



Welcome to our Home Page!

Contact us

Name	St...	Type	Initiator	Size	Time	Waterfall
localhost	200	document	Other	(from ...	3 ms	
favicon.ico	200	x-icon	Other	36.6 KB	3 ms	

2 requests | 36.6 KB transferred | Finish: 225 ms | DOMContentLoaded: 196 ms | Load...

Browser Finickiness and favicon.ico

Browsers are finicky about requesting `favicon.ico`. If they think they already have the icon for a particular domain, they may not request it again, even on a hard refresh. If the above isn't working for you, you can take the whole thing on faith, or you can read `How to Force a Refresh of favicon.ico (https://www.webucator.com/article/how-to-force-a-refresh-of-faviconico/)` for instructions on how to make Google Chrome cooperate.

5.9. Express - Node.js Web Application Framework

As you might imagine, creating all of our web pages in variables like we did in the last example would get pretty cumbersome. Luckily, there is a special Node.js web application framework called Express[10] that makes creating web applications cleaner and easier.

To use Express in a package, `express.js` must be included in the `package.json` dependencies. You can either add it to the file manually or add it by running:

```
npm install express
```

Permissions and Macs

If you are using a Mac and find you get a permissions error when running an `npm install…` command, you may need to run as administrator. To do so, run `sudo npm install…` instead. For example:

```
sudo npm install express
```

Open `Node/Demos/minimum-express/package.json` in your editor. Notice that it currently does not include any dependencies:

Demo 5.8: Node/Demos/minimum-express/package.json

```
{
  "name": "minimum-express",
  "description": "Minimum web app using express",
  "version": "1.0.0"
}
```

Now navigate to `Node/Demos/minimum-express` at the prompt and run:

```
npm install express
```

You may get some warnings, but should not get any errors.

10. https://www.npmjs.com/package/express

Now look again at `Node/Demos/minimum-express/package.json`. It should look something like this, where `n.n.n` is replaced with the latest stable version:

```
{
  "name": "minimum-express",
  "description": "Minimum web app using express",
  "version": "1.0.0",
  "dependencies": {
    "express": "^n.n.n"
  }
}
```

You will also see a new file, `package-lock.json` and a new folder, `node_modules`, added to the `minimum-express` directory:

- `package-lock.json` - used to ensure that future installs are compatible.
- `node_modules` - contains all the modules required to run the application.

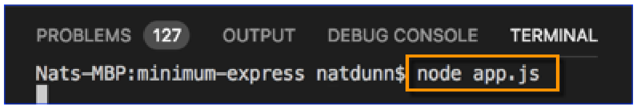
Finally, you will notice we have not included a `server.js` file. Instead, we included an `app.js` file. This is the conventional file name used as the starting point for Express web applications.

Now, recall that `npm start` looks for a "start" script and, if it doesn't find one, defaults to trying to start `server.js`. As such, if we want to start our app using `npm start` we need to provide a "start" script in `package.json` to explicitly tell npm to start `app.js`. We could modify `package.json` to include that:

```
{
  "name": "minimum-express",
  "description": "Minimum web app using express",
  "version": "1.0.0",
  "dependencies": {
    "express": "^n.n.n"
  },
  "scripts": {
    "start": "node app.js"
  }
}
```

This way, we can run the app using `npm start`.

However, we don’t need to use `npm start` to run `app.js`. We can execute `node app.js` directly to run the application:



```
Nats-MBP:minimum-express natdunn$ node app.js
```

❖ 5.9.1. app.js

So, let’s take a look at `app.js` as this is where the magic happens:

Demo 5.9: Node/Demos/minimum-express/app.js

```
const express = require('express');
const app = express();

const html = `<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<title>Hello, world!</title>
</head>
<body>
   <h1>Hello, world!</h1>
</body>
</html>`;

app.get('/', (request, response) => {
    response.status(200);
    response.send(html);
});

app.listen(8080);
```

Code Explanation

Things to notice:

1. The first two lines load in the `express` module and use it to create the `app` object.[11]
2. We then create our `html` variable to hold the HTML code of our Hello-world page.

11. Technically, both express and app are functions, and functions in JavaScript are first-class objects, so they can have their own methods.

3. We then use the `get()` method of the `app` object to handle routing.
    A. The first argument is the path that is requested by the client.
    B. The second argument is a callback function that gets called as a result of this request. Our function simply responds with the status, which will be sent with the header, and the HTML stored in our `html` variable, which will be sent as the response body.
4. Finally, we use `app.listen()` to start listening for requests on port 8080.

5.10. Favicon Middleware

Earlier, we showed how to serve up `favicon.ico` without using Express. Express makes it much easier with the help of Favicon middleware[12].

1. Navigate to `Node/Demos/favicon-express` at the prompt.
2. Install `express` using:

    ```
    npm install express
    ```

    This will add **express** to the dependencies in `package.json` file.
3. Install `express-favicon` using:

    ```
    npm install express-favicon
    ```

    This will add **express-favicon** to the dependencies in `package.json` file.
4. Now run the application:

    ```
    node app.js
    ```

5. Navigate to `http://localhost:8080` in your browser. The `favicon.ico` should show up on the browser tab.

12. https://www.npmjs.com/package/express-favicon

Take a look at the `app.js` file:

Demo 5.10: Node/Demos/favicon-express/app.js

```
const express = require('express');
const app = express();
const favicon = require('express-favicon');

const html = `<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<title>Hello, world!</title>
</head>
<body>
   <h1>Hello, world!</h1>
</body>
</html>`;

const faviconPath = __dirname + '/favicon.ico';
console.log(faviconPath);
app.use(favicon(faviconPath));

app.get('/', (request, response) => {
    response.status(200);
    response.send(html);
});

app.listen(8080);
```

Code Explanation

Things to notice:

1. We require the `express-favicon` module and assign it to the `favicon` constant.
2. We assign the location of `favicon.ico` to the `faviconPath` variable. Note that the special `__dirname` variable contains the resolved absolute path of the directory containing the current file. There is also a special `__filename` variable that contains the resolved absolute path to the current file.
3. We simply pass `favicon(faviconPath)` to `app.use()`. The `app.use()` method requires a middleware function as a callback. We don't have to know the innards of this middleware

function; we only have to know what it does. In this case, the `favicon()` function takes care of sending `favicon.ico` when it is requested.

What is Middleware?

A web server's job is to receive a request and return a response. In the middle, it can perform any number of tasks. Node developers have written *middleware functions*, functions that happen after request and before the response, to make it easier to perform many of the common tasks. We can invoke these functions using `app.use()`.

5.11. Static Files

Saving the full text of different HTML pages in variables as we have been doing is more than a bit unwieldy. Here we will learn how to use Express to easily deliver static files.

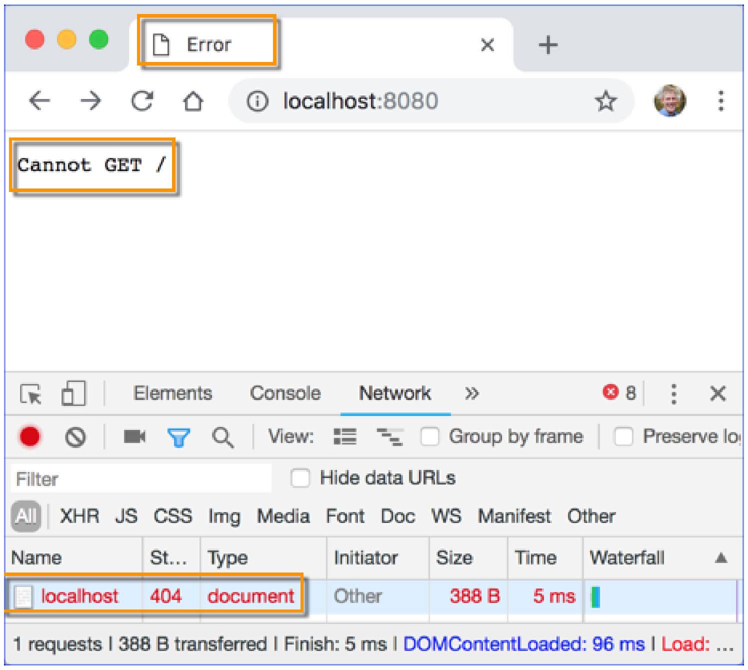
1. Navigate to `Node/Demos/static` at the prompt.
2. Install `express` using:

   ```
   npm install express
   ```

   This will add **express** to the dependencies in `package.json` file.
3. Run the application:

   ```
   node app.js
   ```

4. Navigate to `http://localhost:8080` in your browser. You should see an error:



```
Cannot GET /
```

Name	St...	Type	Initiator	Size	Time	Waterfall
localhost	404	document	Other	388 B	5 ms	

1 requests | 388 B transferred | Finish: 5 ms | DOMContentLoaded: 96 ms | Load: ...

You get this error because we haven't used `app.get('/')` to respond when the root of the site is visited. You will fix that in a moment. But first, let's take a look at the `app.js` file:

Demo 5.11: Node/Demos/static/app.js

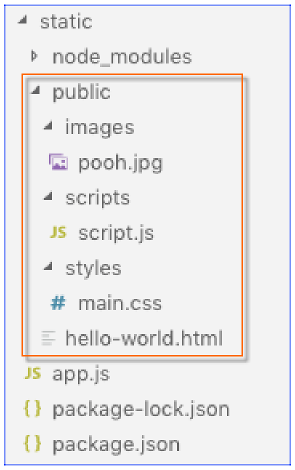
```
const express = require('express');
const app = express();

app.use(express.static('public'));

app.listen(8080);
```

Code Explanation

Notice that it just takes one line of code to specify a directory in which to find all static files:

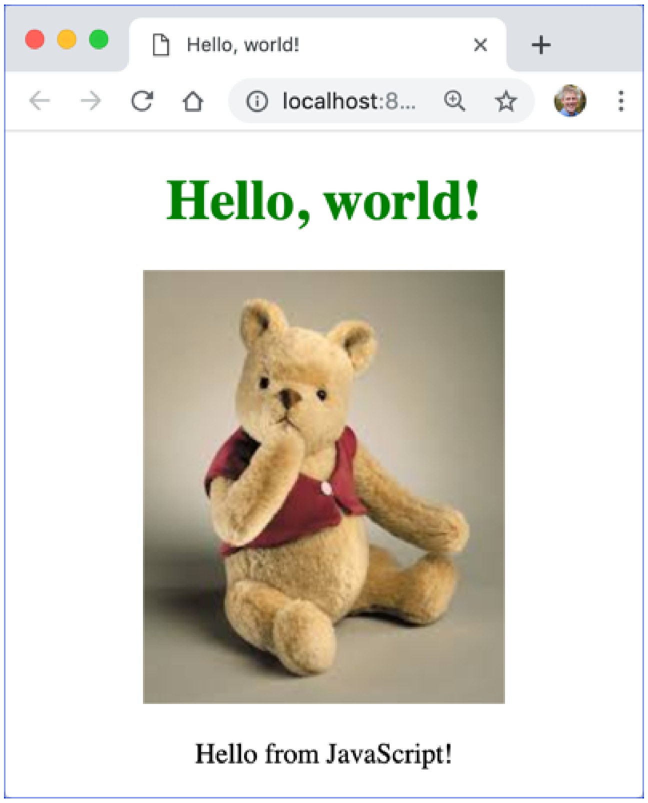
```
app.use(express.static('public'));
```

We pass `express.static('public')` to `app.use` to instruct the application to first look in the `public` folder for a static file to deliver at the matching path. The `public` folder has the following structure:



```
static
  node_modules
  public
    images
      pooh.jpg
    scripts
      script.js
    styles
      main.css
    hello-world.html
  app.js
  package-lock.json
  package.json
```

Now navigate to `http://localhost:8080/hello-world.html`. You should see the following page:



Hello, world!

Hello from JavaScript!

Let’s take a look at that HTML page:

Demo 5.12: Node/Demos/static/public/hello-world.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="styles/main.css">
<script src="scripts/script.js"></script>
<title>Hello, world!</title>
</head>
<body>
  <h1>Hello, world!</h1>
  <main>
  <img src="images/pooh.jpg" alt="Winnie the Pooh">
  <p id="para"></p>
  </main>
</body>
</html>
```

Code Explanation

Notice the files this HTML page references (e.g., `images/pooh.jpg`, `styles/main.css`, and `scripts/script.js`) are also in the `public` folder and, as such, are accessible to the page using relative paths. There is no need to specifically mention any of these files in `app.js`.

Note that there is nothing magic about the name "public." You can call the directory "static" or "myfiles" or anything you like.

Exercise 8: Responding from the Root

10 to 20 minutes

1. Open `Node/Exercises/static` directory. Notice that it is exactly the same as the previous demo at `Node/Demos/static`
2. Install `express` using:

   ```
   npm install express
   ```

   This will add **express** to the dependencies in `package.json` file.
3. Add code to `app.js` to return a simple welcome page when the user navigates to `http://localhost:8080`. Make sure the welcome page includes a link to `hello-world.html`.
4. Run the application and visit `http://localhost:8080` to see if it works.

Solution: Node/Solutions/static/app.js

```
const express = require('express');
const app = express();

app.use(express.static('public'));

const html = `<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<title>Welcome!</title>
</head>
<body>
   <h1>Welcome!</h1>
   <a href="hello-world.html">See Winnie the Pooh!</a>
</body>
</html>`;

app.get('/', (request, response) => {
    response.status(200);
    response.send(html);
});

app.listen(8080);
```

5.12. Processing a Simple Form

Now let's see how we can use Node.js to process a form.

1. Navigate to `Node/Demos/simple-form` at the prompt.
2. Install `express` using:

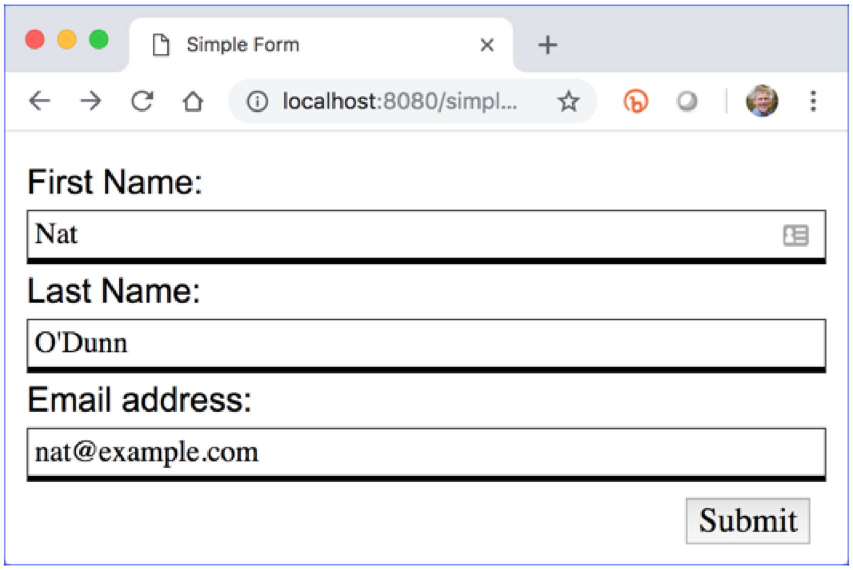
   ```
   npm install express
   ```

   This will add **express** to the dependencies in `package.json` file.

3. Now run the application:

```
node app.js
```

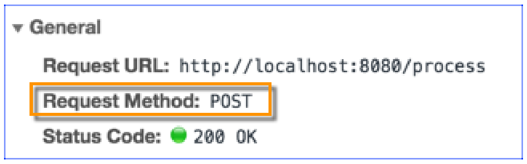
4. Navigate to `http://localhost:8080/simple-form.html` in your browser. You should see a very simple form.
5. Open the Google Chrome Toolbar **Network** tab.
6. Fill in and submit the form. I've intentionally added an apostrophe to my last name to show you how this gets encoded when submitted:



First Name: Nat
Last Name: O'Dunn
Email address: nat@example.com
Submit

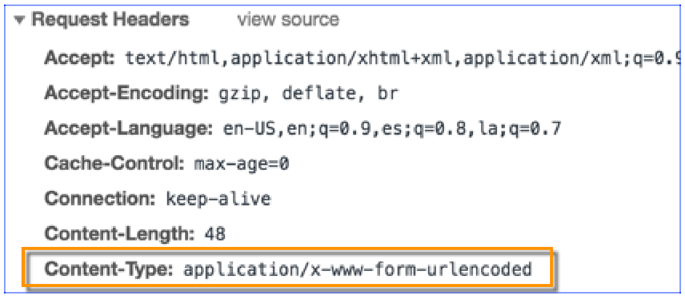
7. Notice in the **Network** tab that the form submitted to `process`. Click `process` and select the **Headers** tab on the right. Notice that:

   A. The **Request Method** is `POST`:



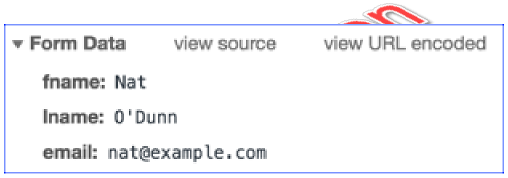
Request URL: http://localhost:8080/process
Request Method: POST
Status Code: 200 OK

   B. The **Request Content-Type** is `application/x-www-form-urlencoded`:



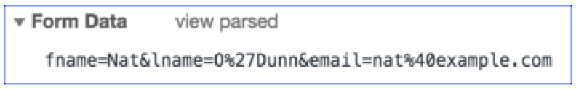
```
▾ Request Headers      view source
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9
    Accept-Encoding: gzip, deflate, br
    Accept-Language: en-US,en;q=0.9,es;q=0.8,la;q=0.7
    Cache-Control: max-age=0
    Connection: keep-alive
    Content-Length: 48
    Content-Type: application/x-www-form-urlencoded
```

C. Click the **Payload** tab. Notice the **Form Data** includes `fname`, `lname`, and `email` and their submitted values:



```
▾ Form Data      view source      view URL encoded
    fname: Nat
    lname: O'Dunn
    email: nat@example.com
```

D. Click the **view source** link next to the **Form Data** heading to see how the data gets received in its raw form:



```
▾ Form Data      view parsed
    fname=Nat&lname=O%27Dunn&email=nat%40example.com
```

This data is URL encoded. It will have to be decoded on the server.

Now, take a look at the `app.js` file:

Demo 5.13: Node/Demos/simple-form/app.js

```
const express = require('express');
const app = express();

app.use(express.static('public'));
app.use(express.urlencoded({ extended: false }));

app.post('/process', (request, response) => {
    const html_process = `<!DOCTYPE html>
    <html lang="en">
    <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Hello, ${request.body.fname}!</title>
    </head>
    <body>
        <h1>Hello, ${request.body.fname}!</h1>
    </body>
    </html>`;
    response.status(200);
    response.send(html_process);
    console.log(request.body);
});

app.listen(8080);
```

Code Explanation

Things to notice:

1. We use the following line to bring in middleware to parse the incoming URL-encoded data:

   ```
   app.use(express.urlencoded({ extended: false }));
   ```

   If our form were used to submit binary data (e.g., to upload an image), we would need to set the `extended` property to `true`.

2. The `express.urlencoded` middleware adds a `body` property to the `request` object. The `body` property is an object that contains key-value pairs generated from the form submission. In our code, we log `request.body` to the console. That will output something like:

   ```
   { fname: 'Nat', lname: 'O\'Dunn', email: 'nat@example.com' }
   ```

We can access each of these properties using dot notation: `request.body.fname`, `request.body.lname`, and `request.body.email`.

3. In this case, we are are just using `request.body.fname` to say hello to the user, but normally we would validate the input and do something more interesting with it.

5.13. Form Validation

Now let's take a look at how we can validate incoming form data. We will use express-validator[13] for this.

1. Navigate to `Node/Demos/form-validation` at the prompt.
2. Install `express` using:

```
npm install express
```

This will add **express** to the dependencies in `package.json` file.

3. Install `express-validator` using:

```
npm install express-validator
```

This will add **express-validator** to the dependencies in `package.json` file.

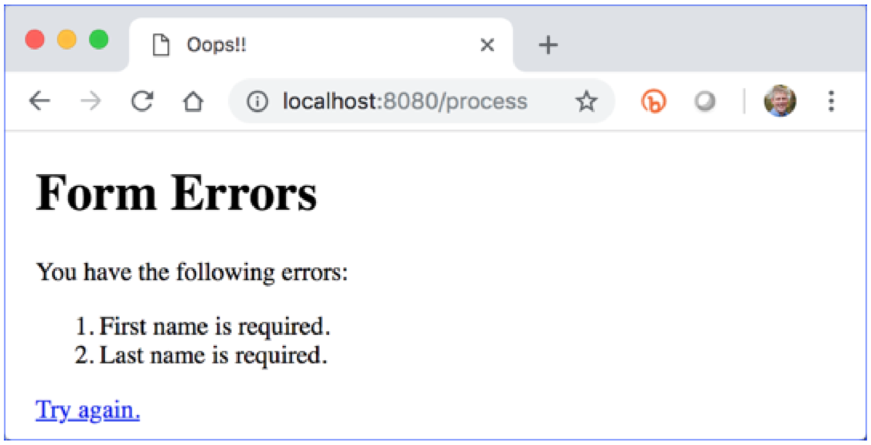
4. Now run the application:

```
node app.js
```

5. Navigate to `http://localhost:8080/simple-form.html` in your browser. You should see the same form we saw earlier. Note that we have no client-side validation on this form as we want to demonstrate what happens when errors get to the server. Normally, however, we would try to force the user to correct as much as possible before submitting the form to the server.

13. https://www.npmjs.com/package/express-validator

6. Submit the form without filling anything out. You should get the following result:



Form Errors

You have the following errors:

1. First name is required.
2. Last name is required.

Try again.

Take a look at the `app.js` file:

Demo 5.14: Node/Demos/form-validation/app.js

```
const express = require('express');
const {check, validationResult} = require('express-validator');
const app = express();


app.use(express.static('public'));
app.use(express.urlencoded({ extended: false }));

app.get('/', (request, response) => {
    response.redirect('/simple-form.html');
});

app.post('/process',
    check('fname','First name is required.').isLength({ min: 1 }),
    check('lname','Last name is required.').isLength({ min: 1 })
, (request, response) => {
    const errors = validationResult(request);
    if (errors.isEmpty()) {
        // code to process form goes here
        response.redirect('/success.html');
    } else {
        let errorList = '';
        for (error of errors.array()) {
            errorList += "<li>" + error.msg + "</li>";
        }
        const html_process = `<!DOCTYPE html>
        <html lang="en">
        <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,initial-scale=1">
        <link rel="stylesheet" href="normalize.css">
        <link rel="stylesheet" href="styles.css">
        <title>Oops!!</title>
        </head>
        <body>
        <main>
            <h1>Form Errors</h1>
            <p>You have the following errors:</p>
            <ol>
                ${errorList}
            </ol>
            <a href='javascript:history.back()'>Try again.</a>
        </main>
        </body>
```

```
            </html>`;
            response.status(200);
            response.send(html_process);
        }

        console.log(request.body);
        console.log(errors.array());
    });

    app.listen(8080);
```

Code Explanation

Things to notice:

1. The following assignment syntax may be new to you:

   ```
   const {check, validationResult} = require('express-validator');
   ```

   This is called *destructuring assignment*. The `require('express-validator')` function returns an object that looks something like this:

   ```
   {
     check: function() {...},
     validationResult: function() {...}
   }
   ```

   Both objects are then deconstructed such that the `check` constant is assigned the `check()` function and the `validationResult` constant is assigned the `validationResult()` function. As a result, this one line of code gives us access to the following two functions:

   A. `check(field, message)` - used to specify what field needs validation and the error message that gets associated with the field if it is invalid. The returned object will take one or more validator methods (e.g., `isLength()`) in a chain.

B. `validationResult(request)` - returns an array of error objects created by the calls to `check()`. For example:

```
[
  {
    location: 'body',
    param: 'fname',
    value: '',
    msg: 'First name is required.'
  },
  {
    location: 'body',
    param: 'lname',
    value: '',
    msg: 'Last name is required.'
  }
]
```

2. Our `app.post()` call now takes a second argument: an array of `check()` calls to validate the fields.
3. In the callback function, we assign the result of the call to `validationResult(request)` to the `errors` constant.
4. We then use an if condition to check to see if the `errors` array is empty:

```
if (errors.isEmpty())
```

   A. If the array **is empty**, we redirect to a success page. We would also normally do some processing with the form variables here. For example, we might enter or modify data in a database.

   B. If the array **is not empty**, we create and send an error page.

5.14. Validators

The **express-validator** middleware uses the validators provided by the validator.js[14] library. The table below shows some of the more common validation methods.

Common Validation Methods from validator.js Library

Validator	Description
`equals(comparison)`	Checks if the value is equal to `comparison`.
`matches(pattern, [modifiers])`	Checks if the value matches a pattern.
`isEmail()`	Checks if the value is a valid email address.
`isURL()`	Checks if the value is a valid URL.
`isEmpty()`	Checks if the value is zero length.
`isIn(valuesArray)`	Checks if the value is found in an array of values.
`isLength(options)`	Checks if the value is between the length of the values set by the `min` (default: `0`) and `max` (default: `undefined`) options.
`isFloat([options])`	Checks if the value represents a float.
`isInt(options)`	Checks if the value represents an integer.
`isNumeric(options)`	Checks if the value represents a number.
`isLowercase()`	Checks if the value is in all lowercase.
`isUppercase()`	Checks if the value is in all uppercase.

There is also a special `exists()` validator that checks to see if a variable exists. Don't confuse this with the `isEmpty()` validator, which checks if the value of a variable is an empty string. The `exists()` validator is often used to see if checkboxes or radio button selections have been made. If the user doesn't check a checkbox or make a radio button selection, then the associated form variable does not get sent to the server. For example, to see if a "terms" checkbox was checked, you could use:

```
check('terms', 'You must accept the terms.').exists()
```

❖ 5.14.1. Validation Chaining

It is possible to check multiple conditions by chaining validation methods. For example, if you want to check if a username is between 8 and 15 characters and is all lowercase, you can do it like this:

14. https://github.com/chriso/validator.js#validators

```
check('username', 'Invalid username.').isLength({ min: 8, max: 15 }).isLowercase()
```

It can be easier to read chained validators when they are separated with new lines like this:

```
check('username', 'Invalid username.').isLength({ min: 8, max: 15 })
  .isLowercase()
```

❖ 5.14.2. not()

The `not()` method is used to negate the following validator. For example, we saw earlier that you could require a non-zero length value using:

```
check('fname', 'First name is required.').isLength({ min: 1 })
```

You could do the same thing by checking that the value is **not empty** using chaining:

```
check('fname', 'First name is required.').not().isEmpty()
```

❖ 5.14.3. withMessage(message)

In some cases, you will want the error message to be specific to the type of error. For example, when we checked to make sure the `username` was between 8 and 15 characters and all lowercase letters, we set the error message to "Invalid username.". It would be more helpful to let the user know what was invalid about it.

The `withMessage(message)` method sets an error message specific to the previous validator in the chain.

```
check('username').isLength({ min: 8, max: 15 })
  .withMessage('Username must be between 8 and 15 characters.')
  .isLowercase()
  .withMessage('Username must be in all lowercase letters.')
```

❖ 5.14.4. Custom Validators

Sometimes the built-in validators don't meet your specific needs. Luckily, express-validator provides a method for creating your own custom validators. The syntax is as follows:

```
check(field, errorMsg).custom(
  (value) => {
    // Code to check if value is valid and
    // return true or false
  }
)
```

For example, let's say that you want the email to be optional, but if it is included, it should be a valid email address. You can do this with the following custom validator:

```
check('email', 'Email must be valid.').custom(
  (value) => {
    const reEmail = /([\w\-\.])+@([\w\-\.])+\.([A-Za-z]{2,4})/;
    return value.length === 0 || reEmail.test(value);
  }
)
```

Try this out by opening the prompt at `Node/Demos/form-validation/` and running `node app-custom.js` at the prompt.

With these methods, you are now ready to do some more advanced form validation.

Exercise 9: Form Validation

20 to 30 minutes

1. Navigate to `Node/Exercises/form-validation` at the prompt.
2. Use npm to install `express` and `express-validator`.
3. Open `Node/Exercises/form-validation/public/ice-cream.html`
4. Open `Node/Exercises/form-validation/app.js` in your editor.
5. Write code to validate the following fields:
    A. `username` - should be between 8 and 25 chars.
    B. `email` - should be a valid email.
    C. `phone` - should use format: `###-###-####`.
    D. `flavor` - should not equal '0'.
    E. `container` - should exist.
    F. `terms` - should exist.

Challenge

Write a custom validator for the `requests` textarea field. If its length is 0 or between 10 and 200, it is valid; otherwise, it is invalid.

Solution: Node/Solutions/form-validation/app.js

```
-------Lines 1 through 11 Omitted-------
app.post('/process', [
    check('username',
        'Username must be 8 to 25 characters.').isLength(
            { min: 8, max: 25 }
        ),
    check('email', 'Invalid Email').isEmail(),
    check('phone', 'Invalid Phone. Use format: ###-###-####').matches(
        /[1-9]\d{2}-\d{3}-\d{4}/
    ),
    check('flavor','Please select a flavor.').not().equals('0'),
    check('container','Please select a container.').exists(),
    check('terms','You must accept the terms.').exists()
], (request, response) => {
-------Lines 25 through 60 Omitted-------
```

Challenge Solution: Node/Solutions/form-validation/app-challenge.js

```
-------Lines 1 through 11 Omitted-------
app.post('/process', [
    check('username',
        'Username must be 8 to 25 characters.').isLength(
            { min: 8, max: 25 }
        ),
    check('email', 'Invalid Email').isEmail(),
    check('phone', 'Invalid Phone. Use format: ###-###-####').matches(
        /[1-9]\d{2}-\d{3}-\d{4}/
    ),
    check('flavor','Please select a flavor.').not().equals('0'),
    check('container','Please select a container.').exists(),
    check('terms','You must accept the terms.').exists(),
    check('requests','Comment must be from 10 to 200 chars.').custom(
        (value) => {
            return value.length === 0 ||
                (value.length >= 10 && value.length <= 200);
        }
    )
], (request, response) => {
-------Lines 31 through 66 Omitted-------
```

5.15. Ajax

❖ 5.15.1. XMLHttpRequest

The `XMLHttpRequest` API allows us to transfer data between client and server - an easy way to fetch data from a URL without a page refresh. Often referred to as Ajax, the `XMLHttpRequest` API lets us update part of a page in response to user actions, submit a form, request a lookup, etc., all without performing a full HTTP request/response to reload the page.

The easiest way to understand how Ajax works is to test it out. First of all, let's take a look at the `app.js` file:

Demo 5.15: Node/Demos/xhr/app.js

```
const express = require('express');
const {check, validationResult} = require('express-validator');
const app = express();

app.use(express.static('public'));
app.use(express.urlencoded({ extended: false }));

app.get('/', (request, response) => {
    response.redirect('/quiz.html');
});

app.post('/process', [
    check('answer').equals('42')
], (request, response) => {
    const errors = validationResult(request);
    console.log(errors.array());
    let msg;
    if (errors.isEmpty()) {
        msg = "<span class='valid'>Right on! " +
            "Did you bring your towel?</span>";
    } else {
        msg = "<span class='invalid'>That is not right, " +
            "but don't panic. You'd far rather be happy " +
            "than right any day.</span>";
    }
    response.status(200);
    response.send(msg);
});

app.listen(8080);
```

Code Explanation

The biggest thing to notice here is that there is nothing particularly new. The only novel thing is that in the `app.post()`, we are not returning a complete HTML page, but just a single `span` element.

Let's see what happens when we submit to this page using the standard HTML form shown below:

Demo 5.16: Node/Demos/xhr/public/quiz-no-ajax.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="normalize.css">
<link rel="stylesheet" href="styles.css">
<title>The ultimate question of life</title>
</head>
<body>
  <form method="post" action="process">
    <label for="answer">
      What is the answer to the ultimate question of
      life, the universe and everything?
    </label>
    <input name="answer" id="answer">
    <button type="submit" id="btn">Check</button>
  </form>
</body>
</html>
```

1. Navigate to `Node/Demos/xhr` at the prompt.
2. Install `express` using:

   ```
   npm install express
   ```

   This will add **express** to the dependencies in `package.json` file.
3. Install `express-validator` using:

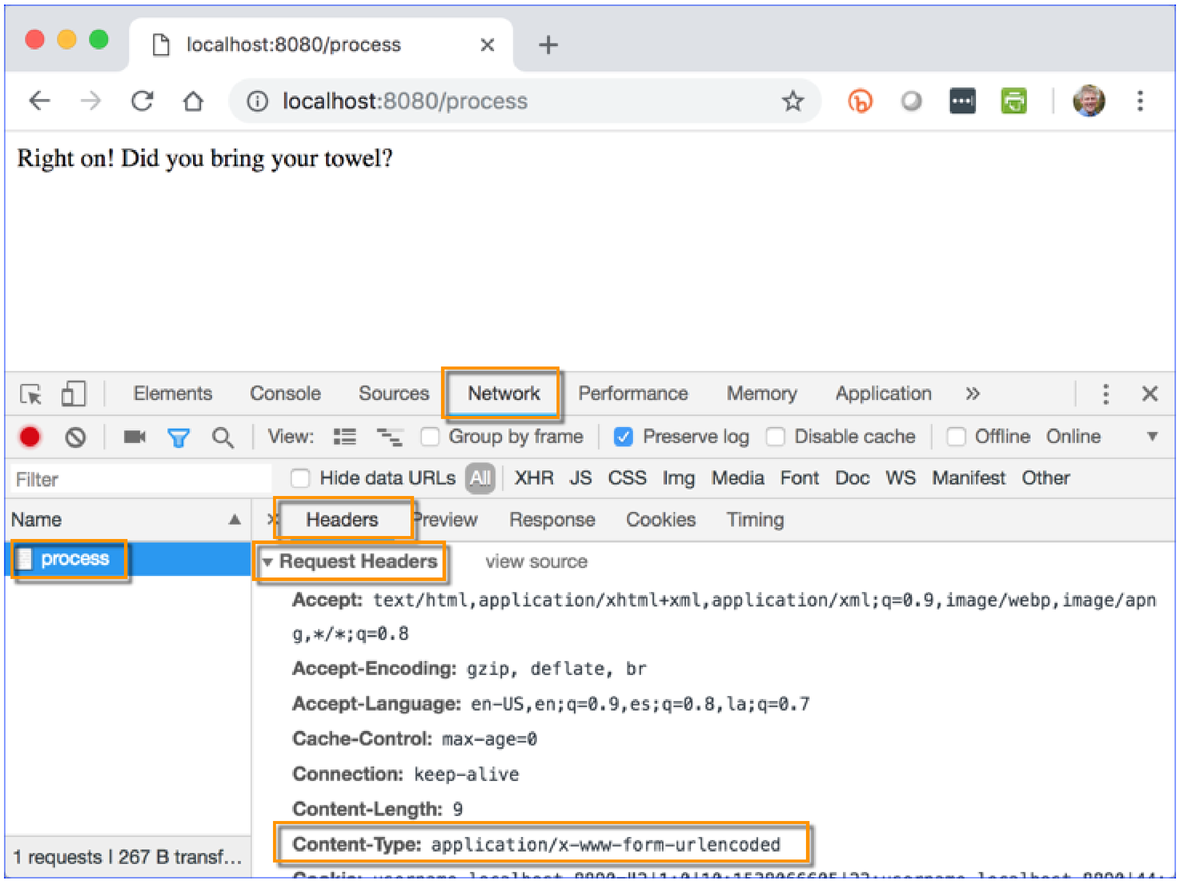
   ```
   npm install express-validator
   ```

   This will add **express-validator** to the dependencies in `package.json` file.
4. Now run the application:

   ```
   node app.js
   ```

5. Now point your browser to `http://localhost:8080/quiz-no-ajax.html`.
6. With Chrome DevTools **Network** tab open, fill out the form (the correct answer is 42), and submit it.

7. Click **process** in the bottom left of Chrome DevTools and, with the **Headers** tab selected, scroll down to the **Request Headers** section:

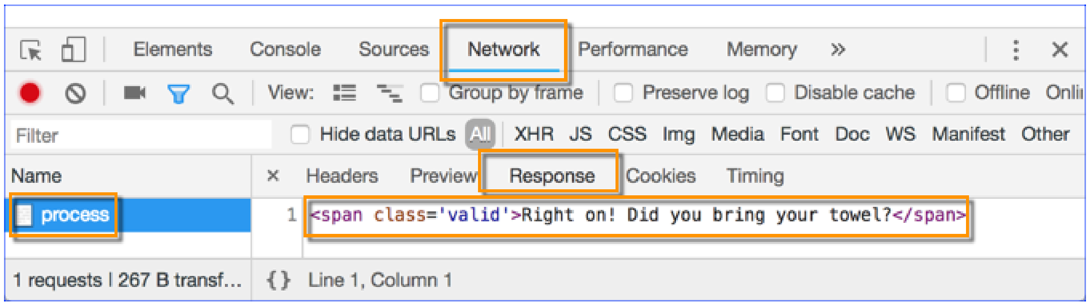


Right on! Did you bring your towel?

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,es;q=0.8,la;q=0.7
Cache-Control: max-age=0
Connection: keep-alive
Content-Length: 9
Content-Type: application/x-www-form-urlencoded
```

Notice the **Content-Type** is 'application/x-www-form-urlencoded'. This is important! **When a form with the method set to `POST` is submitted, the browser sets the `Request`'s `Content-Type` to 'application/x-www-form-urlencoded'.**

8. Click the **Response** tab to see the HTTP response:



```
<span class='valid'>Right on! Did you bring your towel?</span>
```

9. Notice that the full response is just the `span` element. That is because that is all our `app.post()` method returns.
10. Go back to the form and resubmit with an incorrect answer to see the "That's not right..." response returned.

Now take a look at this HTML page, which is almost identical to the last one:

Demo 5.17: Node/Demos/xhr/public/quiz.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="normalize.css">
<link rel="stylesheet" href="styles.css">
<script src="ajax.js"></script>
<title>The ultimate question of life</title>
</head>
<body>
  <form method="post">
    <label for="answer">
      What is the answer to the ultimate question of
      life, the universe and everything?
    </label>
    <input name="answer" id="answer">
    <button type="button" id="btn">Check</button>
    <output id="message"></output>
  </form>
</body>
</html>
```

Code Explanation

Things to notice:

1. We include a new `script` element:

   ```
   <script src="ajax.js"></script>
   ```

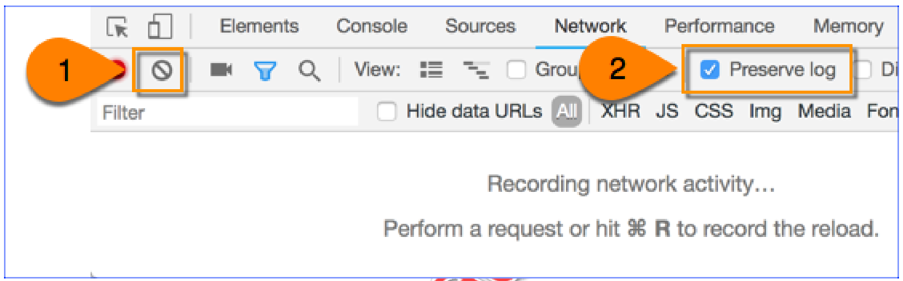
2. The `button` is of type "button" instead of type "submit".

3. There is a new `output` element with the `id` “message”:

```
<output id="message"></output>
```

Now let’s write some Ajax code in the Chrome DevTools Console:

1. Navigate to `http://localhost:8080/` in your browser. You will notice that you get redirected to `http://localhost:8080/quiz.html`
2. Open the Chrome DevTools **Network** tab, clear it, and check **Preserve log**:



Recording network activity…

Perform a request or hit ⌘ R to record the reload.

3. Switch to the Console tab and do the following:
   A. Create a new `XMLHttpRequest` object:

   ```
   const xmlhttp = new XMLHttpRequest();
   ```

   B. Initialize the new `XMLHttpRequest`:

   ```
   xmlhttp.open("POST", '/process', true);
   ```

   C. Set the “Content-type” request header to ’application/x-www-form-urlencoded’. Remember, when *posting* data (i.e., sending data with the POST method), the Request’s Content-Type must be ’application/x-www-form-urlencoded’. When a form is submit the standard HTML way, this gets done automatically, but when we use Ajax to send the data, we must set the `Content-Type`[15] with JavaScript.

   ```
   xmlhttp.setRequestHeader('Content-Type', 'application/x-www-form-urlen ↵
   coded');
   ```

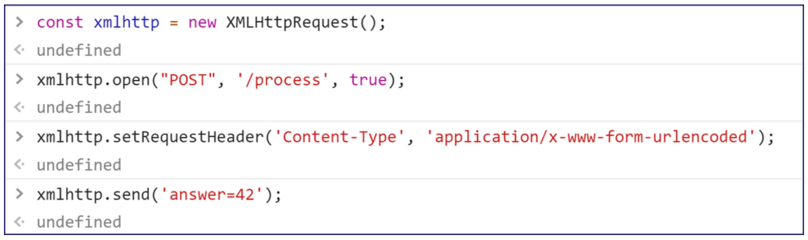
---

15. If you are sending binary data (e.g., uploading an image), then you set the `Content-Type` to “multipart/form-data”.

D. Send the request to the server:

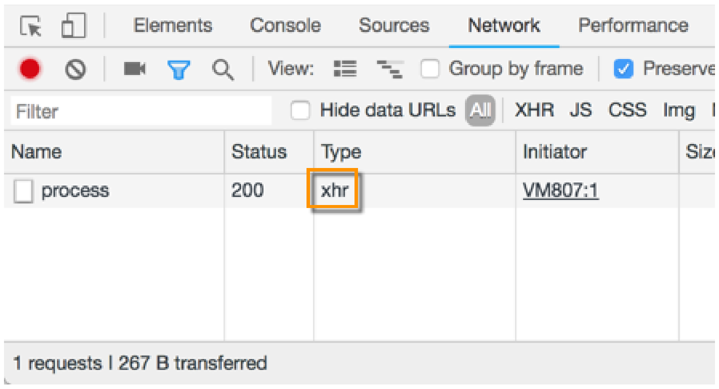
```
xmlhttp.send('answer=42');
```

4. The console should now look like this:



```
> const xmlhttp = new XMLHttpRequest();
<· undefined
> xmlhttp.open("POST", '/process', true);
<· undefined
> xmlhttp.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
<· undefined
> xmlhttp.send('answer=42');
<· undefined
```

5. Now switch to the **Network** tab. You should see this:

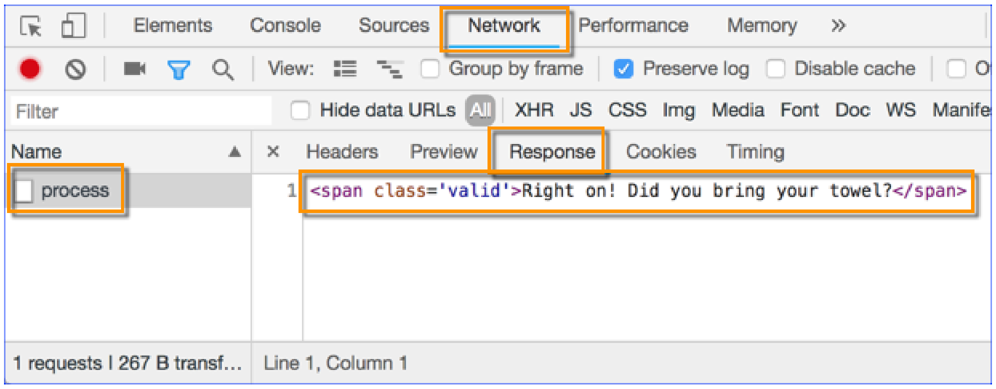


Name	Status	Type	Initiator
process	200	xhr	VM807:1

1 requests | 267 B transferred

Notice that the type is `xhr` for XMLHttpRequest.

6. Now click **process** and look at the **Response** tab. You see that the same `span` element we saw earlier is returned:



```
<span class='valid'>Right on! Did you bring your towel?</span>
```

1 requests | 267 B transf…   Line 1, Column 1

The GET Method

This example uses the POST method to send the data to the server. You can also send data using the GET method. When using the GET method, you pass the parameters on the querystring, like this:

```
xmlhttp.open("GET", '/process?answer=42', true);
```

In this case, there is no need to call `xmlhttp.setRequestHeader()` to change the Content-Type.

There is also no need to pass data in the call to `xmlhttp.send()`, so you pass `null` instead:

```
xmlhttp.send(null);
```

❖ 5.15.2. Asynchronous

The "A" in Ajax stands for **A**synchronous. In this context, asynchronous means that the process runs independently of other processes. This allows the web page to continue to respond to events while it waits for a response from the server to the `XMLHttpRequest`. The server sends several responses back to the client updating it on the current status. The `readyState` property of the `XMLHttpRequest` changes as a result of these responses. Possible `readyState` values are shown in the table below:

XMLHttpRequest readyState Values

Value	State	Description
0	UNSENT	`XMLHttpRequest` created but not opened.
1	OPENED	`XMLHttpRequest` opened.
2	HEADERS_RECEIVED	`XMLHttpRequest` sent and headers have been received.
3	LOADING	Downloading. The response text has begun to download.
4	DONE	The full response has been received.

Generally, when a request is made to the server using Ajax, we will want to do something with the server's response. As we don't know when the response will be complete, we have to write code to listen for it. This is done using the `onreadystatechange` event handler:

```
xmlhttp.onreadystatechange = function() {
  // do something
}
```

But the above code has a problem. It will run the function code every time the `readyState` changes. We usually only want to run the code when the full response has been recieved and has a status of 200, meaning all is good. So, we need an if condition:

```
xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
    // do something
  }
}
```

Now it's time to include this code in our web application. We will do this in the `ajax.js` file, which is included in our `quiz.html` file with this line of code:

```
<script src="ajax.js"></script>
```

Demo 5.18: Node/Demos/xhr/public/ajax.js

```
function checkAnswer(e) {
  const answer = document.getElementById('answer').value;
  const xmlhttp = new XMLHttpRequest();
  const output = document.getElementById("message");
  output.innerHTML = "validating...";
  xmlhttp.open("POST", '/process', true);
  xmlhttp.onreadystatechange = function() {
    console.log(xmlhttp.readyState);
    if (xmlhttp.readyState == XMLHttpRequest.DONE
        && xmlhttp.status == 200) {
      output.innerHTML = xmlhttp.responseText;
    }
  }
  xmlhttp.setRequestHeader('Content-Type',
    'application/x-www-form-urlencoded');
  xmlhttp.send('answer=' + answer);
}

window.addEventListener('load', function(e) {
  const btn = document.getElementById('btn');
  const answer = document.getElementById('answer');
  btn.addEventListener('click', checkAnswer);

  answer.addEventListener('keydown', function(e) {
    if (e.key === 'Enter') {
      checkAnswer(e);
      e.preventDefault();
    }
  });
});
```

Code Explanation

This code should make sense as it is mostly the same as the code we wrote in Chrome DevTools Console. The only changes and additions are:

1. We use the `XMLHttpRequest.DONE` constant instead of the number 4 to check the `readyState`. That's just a constant holding value 4.
2. We set the `innerHTML` of the `output` element to `xmlhttp.responseText`, which is the response text returned from the server.
3. We also log `xmlhttp.readyState`. Check out the Chrome Console after checking the answer to see the results of that.

Avoid Synchronous XMLHttpRequests

It is possible to make a *synchronous* XMLHttpRequest by setting the third parameter of the `xmlhttp.open()` method to `false`.

```
xmlhttp.open("POST", '/process', false);
```

However, you should avoid doing this because it can be disruptive to the user (the page can “freeze” as it waits for a process to finish).

The only reason we point this out at all is to explain what that third parameter of the `open()` method is: `true`: asynchronous, `false`: synchronous.

Exercise 10: Form Validation with Ajax

20 to 30 minutes

1. Examine the application in `Node/Exercises/xhr`.
2. Open `Node/Exercises/xhr/app.js` in your editor. Add an end-point (i.e., a route) for a POST request to '/coupon-check' that responds with one of the following:

   ```
   '<span class="valid">Coupon validated</span>'
   ```

   ```
   '<span class="invalid">Invalid coupon</span>'
   ```

   It should check to see that the passed-in coupon code is in the `couponCodes` array defined above.
3. Open `Node/Exercises/xhr/public/ajax.js`. Write the code to post the coupon code to '/coupon-check' using Ajax. The parameter should be `coupon` and the value should be `couponCode` (as defined in the beginning of the function).
4. Navigate to `Node/Exercises/xhr` at the prompt.
5. Use npm to install `express` and `express-validator`.
6. Now run the application:

   ```
   node app.js
   ```

   In your browser, navigate to `http://localhost:8080/ice-cream.html`, and test the coupon code functionality. Use Chrome DevTools **Console** and **Network** tabs to debug any issues that come up.

Solution: Node/Solutions/xhr/app.js

```
-------Lines 1 through 17 Omitted-------
app.post('/coupon-check', [
    check('coupon', 'Invalid coupon code').isIn(couponCodes)
], function(request, response) {
    const errors = validationResult(request);

    let msg;
    if (errors.isEmpty()) {
        msg = '<span class="valid">Coupon validated</span>';
    } else {
        msg = '<span class="invalid">Invalid coupon</span>';
    }
    response.status(200);
    response.send(msg);
});
-------Lines 32 through 93 Omitted-------
```

Solution: Node/Solutions/xhr/public/ajax.js

```
function checkCouponCode(e) {
  const couponCode = coupon.value;
  const xmlhttp = new XMLHttpRequest();
  const output = document.getElementById("coupon-message");
  output.innerHTML = "validating...";
  xmlhttp.open("POST", '/coupon-check', true);
  xmlhttp.onreadystatechange = function() {
    console.log(xmlhttp.readyState);
    if (xmlhttp.readyState == XMLHttpRequest.DONE
        && xmlhttp.status == 200) {
      output.innerHTML = xmlhttp.responseText;
    }
  }
  xmlhttp.setRequestHeader('Content-Type',
    'application/x-www-form-urlencoded;charset=UTF-8');
  xmlhttp.send('coupon=' + couponCode);
}


window.addEventListener('load', function(e) {
  const btnCoupon = document.getElementById('btn-coupon');
  const coupon = document.getElementById('coupon');
  coupon.autocomplete = 'nope';
  btnCoupon.addEventListener('click', checkCouponCode);

  coupon.addEventListener('keydown', function(e) {
    if (e.key === 'Enter') {
      checkCouponCode(e);
      e.preventDefault();
    }
    console.log(e.key);
  });
});
```

Conclusion

In this lesson, you have learned:

- To install and use Node.js.
- To do server-side validation of forms.

LESSON 6
JSON

Topics Covered

☑ Array and object literals.

☑ JSON for data transfer in JavaScript

Introduction

JSON, or JavaScript Object Notation, is a data-interchange format commonly used in JavaScript.

6.1. JSON

JSON is a lightweight format for exchanging data between the client and server. It is often used in Ajax applications because of its simplicity and because its format is based on JavaScript object literals, which are textual representations of objects. We will start by reviewing JavaScript's object-literal syntax and then we will see how we can use JSON in an Ajax application.

6.2. Review of Object Literals

❖ 6.2.1. Arrays

Array literals are created with square brackets as shown below:

```
const beatles = ["Paul", "John", "George", "Ringo"];
```

This is the equivalent of:

```
const beatles = new Array("Paul", "John", "George", "Ringo");
```

❖ 6.2.2. Objects

Object literals are created with curly brackets:

```
const beatles = {
  "Country" : "England",
  "YearFormed" : 1959,
  "Style" : "Rock'n'Roll"
}
```

This is the equivalent of:

```
const beatles = new Object();
beatles.Country = "England";
beatles.YearFormed = 1959;
beatles.Style = "Rock'n'Roll";
```

Just as with all objects in JavaScript, the properties can be references using dot notation or bracket notation.

```
beatles.Style; //Dot Notation
beatles["Style"]; //Bracket Notation
```

❖ 6.2.3. Arrays in Objects

Object literals can contain array literals:

```
const beatles = {
  "Country" : "England",
  "YearFormed" : 1959,
  "Style" : "Rock'n'Roll",
  "Members" : ["Paul", "John", "George", "Ringo"]
}
```

❖ 6.2.4. Objects in Arrays

Array literals can contain object literals:

```
const rockbands = [
  {
    "Name" : "Beatles",
    "Country" : "England",
    "YearFormed" : 1959,
    "Style" : "Rock'n'Roll",
    "Members" : ["Paul", "John", "George", "Ringo"]
  },
  {
    "Name" : "Rolling Stones",
    "Country" : "England",
    "YearFormed" : 1962,
    "Style" : "Rock'n'Roll",
    "Members" : ["Mick", "Keith", "Charlie", "Bill"]
  }
]
```

6.3. Back to JSON

On the JSON website (`https://www.json.org`), JSON is described as:

1. “a lightweight data-interchange format”
2. “easy for humans to read and write”
3. “easy for machines to parse and generate”

Numbers 1 and 3 are certainly true. Number 2 depends on the type of human. Experienced programmers will find that they can get comfortable with the syntax relatively quickly.

❖ 6.3.1. JSON Syntax

The JSON syntax is like JavaScript’s object literal syntax except that the objects cannot be assigned to a variable. JSON just represents the data itself. So, the `Beatles` object we saw earlier would be defined as follows:

```
{
  "Name" : "Beatles",
  "Country" : "England",
  "YearFormed" : 1959,
  "Style" : "Rock'n'Roll",
  "Members" : ["Paul", "John", "George", "Ringo"]
}
```

❖ 6.3.2. The built-in JavaScript JSON Object

JavaScript comes with a built-in `JSON` object, which has the following very useful methods:

- `JSON.parse(strJSON)` - converts a JSON string into a JavaScript object.
- `JSON.stringify(objJSON)` - converts a JavaScript object into a JSON string.

The process for sending data between the browser and server with JSON is as follows:

1. On the client-side:
    - Create a JavaScript object using the standard or literal syntax.
    - Use the JSON parser to stringify the object.
    - Send the URL-encoded JSON string to the server as part of the HTTP Request. This can be done using the HEAD, GET or POST method by assigning the JSON string to a variable. It can also be sent as raw text using the POST method, but this may create extra work for you on the server-side.
2. On the server-side:

- Convert the incoming JSON string to an object using a JSON parser for the language of your choice. Many languages have built-in JSON parsers. The methods available depend upon which parser you are using. See the parser's documentation for details.
- Do whatever you wish with the object.
- If you wish to send JSON back to the client:
    - Create a new object for storing the response data.
    - Convert the new object to a string using your JSON parser.
    - Send the JSON string back to the client as the response body.

3. On the client-side:
    - Convert the incoming JSON string to an object using the JavaScript JSON parser.
    - Do whatever you wish with the object.
    - And so on…

The following example shows how to transfer data to the server using JSON. The server then sends a plain text response back. Let's look first at the HTML file:

Demo 6.1: JSON/Demos/simple-json/public/send-json.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link rel="stylesheet" href="normalize.css">
<link rel="stylesheet" href="styles.css">
<script src="ajax.js"></script>
<title>Using JSON</title>
</head>
<body>
<main>
  <h1>Request</h1>
  <button id="hi">Hi There!</button>
  <button id="bye">Good bye!</button>
  <h1>Response</h1>
  <output id="response">Waiting...</output>
</main>
</body>
</html>
```

Code Explanation

Just notice that this HTML file includes `ajax.js` and that it has two buttons with ids “hi” and “bye” and an `output` element with the id “response”.

Now let’s look at the server-side `app.js` file:

Demo 6.2: JSON/Demos/simple-json/app.js

```
const express = require('express');
const app = express()
app.use(express.static('public'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));

app.get('/', (request, response) => {
    response.redirect('/send-json.html');
});

app.get('/process', (request, response) => {
  const strJSON = request.query['strJSON'];
  const objJSON = JSON.parse(strJSON);
  let responseMsg;
  if (objJSON.msg === "Hi There!") {
    responseMsg = "And hi there to you!";
  } else {
    responseMsg = "Later Gator!";
  }
  response.send(responseMsg);
});

app.listen(8080);
```

Code Explanation

Things to note:

1. We are using the `app.get()` method to process the form.
2. `request.query` will hold the querystring parameters. The callback function expects a “strJSON” parameter in the querystring and assigns its value to `strJSON`:

    ```
    const strJSON = request.query['strJSON'];
    ```

3. We use `JSON.parse()` to convert `strJSON` to an object and assign that object to `objJSON`.
4. We set the value of `responseMsg` based on the value of the "msg" key of `objJSON` and then we send `responseMsg` back to the client.

And finally, let's look at the client-side `ajax.js` file:

Demo 6.3: JSON/Demos/simple-json/public/ajax.js

```
function sendRequest(msg) {
    document.getElementById("response").innerHTML = "";
    const objJSON = {
        "msg": msg
    };
    const strJSON = encodeURIComponent(JSON.stringify(objJSON));
    const xmlhttp = new XMLHttpRequest();
    xmlhttp.open("get", "process?strJSON=" + strJSON, true);

    const output = document.getElementById("response");
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == XMLHttpRequest.DONE
            && xmlhttp.status == 200) {
            output.innerHTML = xmlhttp.responseText;
        }
    }

    xmlhttp.send(null);
}

window.addEventListener('load', function() {
    const btnHi = document.getElementById("hi");
    const btnBye = document.getElementById("bye");
    btnHi.addEventListener('click', function() {
        sendRequest(btnHi.innerHTML);
    }, false);
    btnBye.addEventListener('click', function() {
        sendRequest(btnBye.innerHTML);
    }, false);
});
```

Code Explanation

Things to note:

1. When the page loads, we create event listeners for clicks on the “hi” and “bye” buttons. The callback functions for both call `sendRequest()` and pass in the `innerHTML` of the button.
2. The `sendRequest()` function does the following:
   A. Creates a simple JSON object with one property holding the passed in `msg` string.
   B. Converts the JSON object to a string and encodes it for passing to the server using the `encodeURIComponent()` method.
   C. Passes the string as a parameter of our Ajax request, which uses the GET method.
   D. Outputs the `responseText` to the page.

Run this app:

1. Navigate to `JSON/Demos/simple-json` at the prompt.
2. Install `express` using:

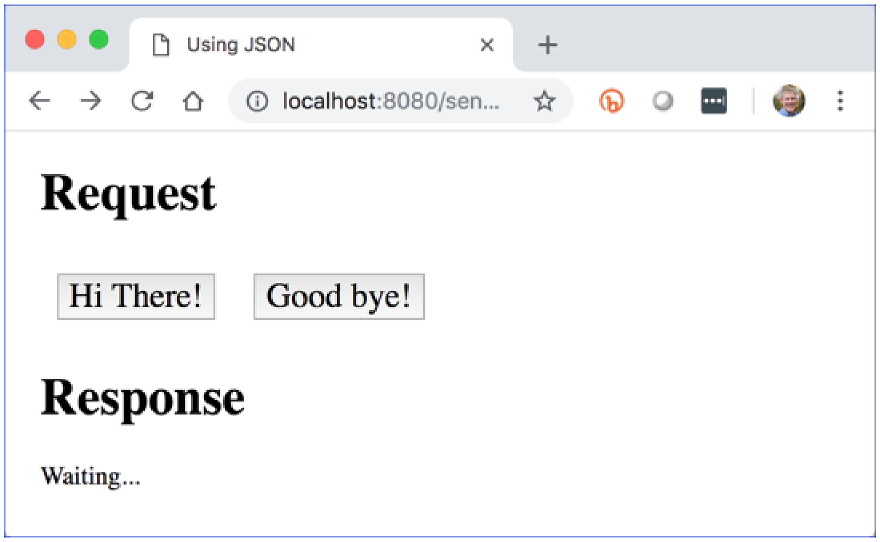
```
npm install express
```

   This will add **express** to the dependencies in `package.json` file.

3. Now run the application:

```
node app.js
```

4. Now point your browser to `http://localhost:8080`. This will redirect to `http://localhost:8080/send-json.html`:



Request

Hi There! Good bye!

Response

Waiting...

5. Press the buttons. The response should change to "And hi there to you!" and "Later Gator!"

Encoding and Decoding URI Components

You should follow this process when sending JSON from the client to the server:

1. Create JavaScript object:

```
const objJSON = { // properties };
```

2. Use `JSON.stringify()` to convert the object to a JSON string:

```
const strJSON = JSON.stringify(objJSON);
```

3. Use `encodeURIComponent()` to encode the string for transmitting to the server:

```
xmlhttp.send("strJSON=" + encodeURIComponent(strJSON));
```

Likewise, you should follow this process when receiving a JSON string from the server:

1. Use `decodeURIComponent()` to decode the string received from the server:

```
const strJSON = decodeURIComponent(xmlhttp.responseText);
```

2. Use `JSON.parse()` to convert the JSON string to an object:

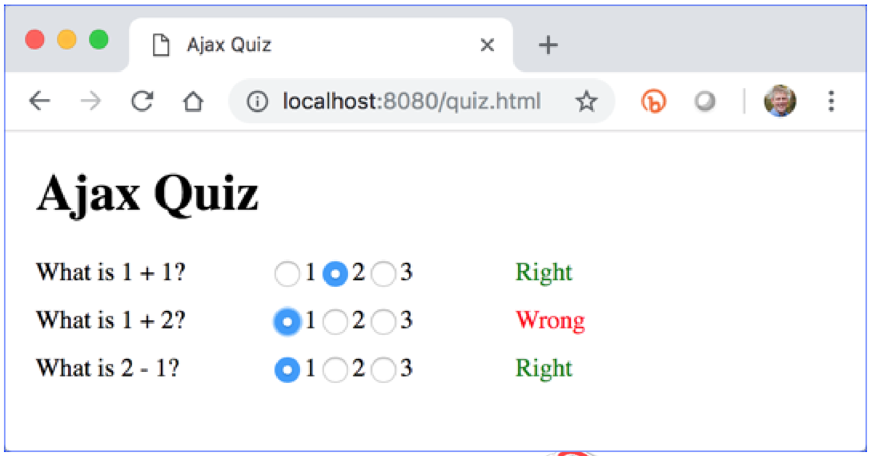
```
const objJSON = JSON.parse(strJSON);
```

3. Do what you need to do with the JavaScript object.

Exercise 11: Using JSON

30 to 40 minutes

In this exercise, you will create an Ajax quiz that uses JSON to pass data to the server. The web page looks like this:



Ajax Quiz

What is 1 + 1?	○ 1 ● 2 ○ 3	Right
What is 1 + 2?	● 1 ○ 2 ○ 3	Wrong
What is 2 - 1?	● 1 ○ 2 ○ 3	Right

1. Examine the application in `JSON/Exercises/quiz`.
2. Open `JSON/Exercises/quiz/app.js` in your editor. This file is already complete. Notice the GET request end-point to '/process-quiz' that:
    A. Gets the value of `strJSON` from the querystring and assigns it to `strJSON`.
    B. Converts `strJSON` to an object using `JSON.parse()` and assigns the result to `objJSON`.
    C. Creates `question` and `answer` variables based on the values of the `objJSON` keys by the same names, which will contain the question number (e.g., `q1`, `q2`, or `q3`) and the user's answer.
    D. Uses a switch-case statement to determine whether the answer is correct and creates an appropriate response message (`responseMsg`).
    E. Sends the response to the client.
3. At the prompt, navigate to the directory `JSON/Exercises/quiz` and install express using npm.
4. Run `node app.js` to start the application.
5. Open your browser and navigate to `http://localhost:8080/process-quiz?strJSON={"question":"q1","answer":2}`.

Notice that it responds with "Right". You will write the Ajax code to send this request, receive the response and output the response without refreshing the page.

6. Open `JSON/Exercises/quiz/public/quiz.html` in your editor and examine the code.
7. Review the HTML. Notice the questions are named "q1", "q2", and "q3" and that each question has an associated result `output` element.
8. Open `JSON/Exercises/quiz/public/ajax.js` in your editor and examine the code.
9. When the page loads, we create event listeners for click events on all the radio buttons with a callback function of `checkAnswer()`.
10. Your job is to write the `checkAnswer()` function. It should do the following:
    A. Create an object with two properties: `question` and `answer`, that hold the question number (e.g, "q3") and the user's answer (e.g, "3").
    B. Pass that object to the JSON parser to stringify. That should result in a string like: `{"question" : "q3", "answer" : "3"}`
    C. Encode that, and store it in a new object as the `strJSON` property.
    D. Write out "checking..." to the appropriate result `div` while awaiting a response from the server.
    E. Using Ajax, send the data to the server to be processed by `/process-quiz`, which expects a `strJSON` parameter to be sent using the GET method.
    F. When the response comes, set the `innerHTML` of the appropriate `output` element to the `xmlhttp.responseText`.
11. Test your solution in a browser by visiting `http://localhost:8080` and responding to the questions.

Challenge

Modify the quiz so that all questions are processed at once:

1. Open `JSON/Exercises/quiz/public/quiz-challenge.html` in your editor and examine the code. Notice that it is similar to `quiz.html`, but the questions are now wrapped in a `<form>` tag and there is a submit button.
2. Open `JSON/Exercises/quiz/app.js` in your editor and examine the post end-point to 'process-challenge'.
3. Open `JSON/Exercises/quiz/public/ajax-challenge.js` in your editor and examine the code, some of which has been written already.

4. Your job is to write the `checkQuiz()` function, which should create a JSON string with the following format:

```
{ "answers" : [a1, a2, a3] }
```

where `a1`, `a2`, and `a3` hold the user's answers, or "x" if the question is not answered. For example, if the user answers 3 for the first question, 2 for the second question, and leaves the third question unanswered, the JSON string should look like this: `{ "answers" : ["1", "3", "x"] }`. The function should then pass that string to the `/quiz-challenge` Node.js response route, which, as we saw above, is already written.

5. The server-side script will return a JSON string with the following format:

```
{
  "q1": "<span class='right'>Right</span>",
  "q2": "<span class='wrong'>Wrong</span>",
  "q3": "<span class='unanswered'>Unanswered</span>"
}
```

6. Based on the string the server returns, write out the server responses to the `q1Result`, `q2Result`, and `q3Result` divs. Note that, to be safe, you should always pass `xmlhttp.responseText` to `decodeURIComponent()` before using the `JSON` object to parse it:

```
const strJSON = decodeURIComponent(xmlhttp.responseText);
const objJSON = JSON.parse(strJSON);
```

7. Test your solution in a browser by visiting `http://localhost:8080/quiz-challenge.html`, responding to the questions, and clicking the "Check Answers" button.

Solution: JSON/Solutions/quiz/public/ajax.js

```
function checkAnswer(e) {
    const target = e.target;
    const q = target.name;
    const a = target.value;
    const objQuestion = {
        question: q,
        answer: a
    };
    const output = document.getElementById(q + "Result");
    const strJSON = encodeURIComponent(JSON.stringify(objQuestion));
    output.innerHTML = "checking...";
    const xmlhttp = new XMLHttpRequest();
    xmlhttp.open("get", "/process-quiz?strJSON=" + strJSON, true);
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4 && xmlhttp.status === 200) {
            const response = decodeURIComponent(xmlhttp.responseText);
            output.innerHTML = response;
        }
    }

    xmlhttp.send(null);
}

window.addEventListener('load', function() {
    const answers = document.querySelectorAll("input[type='radio']");
    for (answer of answers) {
        answer.addEventListener('click', checkAnswer);
    }
})
```

Challenge Solution: JSON/Solutions/quiz/public/ajax-challenge.js

```
function checkQuiz(form) {
    const a1 = getAnswer(form.q1);
    const a2 = getAnswer(form.q2);
    const a3 = getAnswer(form.q3);
    const objJSON = {
        "answers" : [a1, a2, a3]
    }
    const strJSON = JSON.stringify(objJSON);
    const xmlhttp = new XMLHttpRequest();
    xmlhttp.open("post","/process-challenge", true);
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
            const strJSON = decodeURIComponent(xmlhttp.responseText);
            const objJSON = JSON.parse(strJSON);
            for (let i in objJSON) {
                const output = document.getElementById(i + 'Result');
                output.innerHTML = objJSON[i];
            }
        }
    }
    xmlhttp.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded");
    xmlhttp.send("strJSON=" + encodeURIComponent(strJSON));
}
-------Lines 25 through 41 Omitted-------
```

A More Involved Application

If you're interested in seeing a more involved application, check out `JSON/Demos/presidents/`.

Conclusion

In this lesson, you have learned:

- How to use array and object literals.
- What JSON is and how to use it for data transfer in JavaScript.