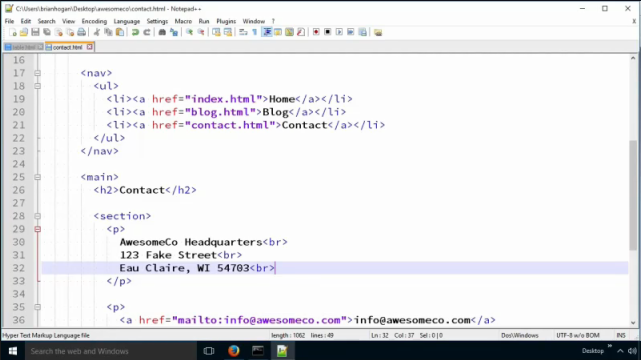
**Tutorial: AwesomeCo – Forms**

Forms let us add interactivity to our web pages. They give us a way to let visitors to our site provide us with information.

Let's add a contact form to the AwesomeCo web site's `contact` page.

**Unzip the latest AwesomeCo files provided for this tutorial.**

**Open the contact.html and style.css page in your text editor.**

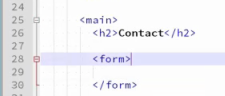


To build a form, we use the `form` element, which has an opening and closing tag.

**Add form tag below the h2 inside the main tag:**

**<form>**

**</form>**



Now, when we want to create a form on a web page, we need to know a couple of things. First, we need to determine where the form data will go.

HTML forms have an `action` attribute that we use to specify the URL of the server-side script or program that will receive our form and process it.

We need a full-blown web server and some server-side programming language like PHP, Perl, Java, C#, Ruby, or NodeJS to process our form. We can't just whip up a simple web page. We submit our form data to this server-side program, and the program processes the data and sends us a new HTML page back telling us if we were successful or not.

Writing a server-side form processing script is beyond the scope of this class. In future classes, you'll learn how to write code that does this. But for this class, I have a server that you'll send your form data to.

**Add the following action attribute shown in bold below to your open form tag:**

<form **action="https://itsd.cvtc.edu/web1/awesomeco.php">**

</form>

This server-side script doesn't really do anything other than let us know if the form submission worked or not. In a real script, we'd have the script send an email to the appropriate person at the company who could address or ignore the message that comes in.

Next, we need to determine the type of request we are going to make to that server. There are two basic types of request that we can do inside of a web browser... GET requests and POST requests, and the one we choose is based on what we want to do.

The type of request is known as the METHOD of the request.

A GET request is usually a request for information. When we click a link, we're making a GET request. When we enter something into the search field on Google, that form uses a GET request too.

A GET request is supposed to be idempotent, which means that each time we issue it it should do exactly the same thing each time. Nothing should change.

However, if we want to make a change to something, we can use a POST request.

When we make a form, we get to choose which method we want to use. If we use a GET request, the data we enter into the form will actually be passed in the URL; we'll actually see the data we typed in.

If we use a POST request, the data we put into the form gets placed into a special place called the Request Header, which is a bunch of data that describes the request. Both GET and POST requests have a request header, but using a POST request puts the form data into the request header.

The rule of thumb is that we use GET requests for "safe" data. That means we use them for things like searches and information requests where we expect the server to tell us something useful. We use the POST request when we have to do something that can't be undone. Something that causes the server to make a change to something.

We're building a form that will send data to a server so the server can record that data and someone can follow up on it. For this scenario, we will use a POST request.

**Add the following method attribute shown in bold below to your open form tag:**

<form **method="post"** action=" **https://itsd.cvtc.edu/web1/awesomeco.php**">

</form>

So that's the form, but now we need to put in some fields. So, let's make a text field that will let the user enter her first name.

To do that we use the `input` tag and we use the `type` attribute. The value of the `type` attribute describes the kind of content we are expecting the user to enter into the field.

**Add the input tag shown in bold below inside the open form tag:**

**<input type="text" name="first\_name">**

We also give it a name attribute. The form that processes the script will be able to figure out what to do with it. The data goes over to the server in what we call a map, which is a collection of names and values.

So, when we send this data over, we're sending over both the name of the field and the data the user entered.

The server-side script will then be able to identify the data.

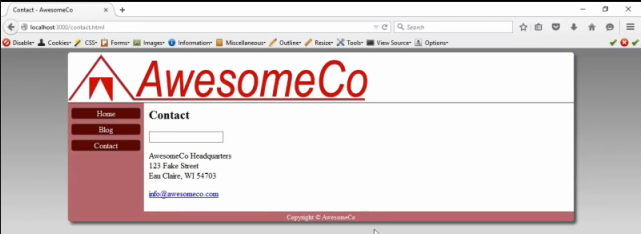
The server-side script you're using determines the names of the form fields on your web page, though. Either you are given a list of the field names, or you have to figure them out by reading code or documentation. Many times, you'll be

writing both the server-side form processor yourself, so you can make the choices you need. But in this class, we will not be doing that.

In this class, I'll tell you the names of the form fields to use. And as long as you use them, your forms will work.

Notice that the `input` tag doesn't have a closing tag? It's a lot like the `img` tag, it is content.

**Open Firefox and run the contact.html page.**



Now when we refresh in the browser we only see a field. We don't know what to do. We need to provide a label for this field, and there's a special way we need to do that. We don't just use a paragraph. We use a `label` element.

**Add the following label tag above the input tag:**

**<label for="first\_name">First name</label>**

The `for` attribute is how we link the label to the field. This helps screen reading software map things together. However, the `for` attribute doesn't point to the field's `name` attribute. The `name` attribute is just something the server-side script uses when it receives the data.

To associate the label to the field, we add an `id` to the field.

**Add the following id attribute shown in bold below to the input tag:**

<input type="text" name="first\_name" **id="first\_name">**

I like to make my IDs and names the same when I code forms.



**Save the contact.html page and refresh the browser**.



We now see the label and the field.

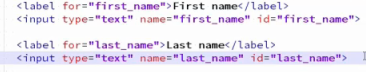
So, let's add the last name field to the form.

To do that we just duplicate what we already have. Use copy and paste and replace first with last name.

**Add the following bolded tags below the last input tag:**

**<label for="last\_nane">Last name</label>**

**<input type="text" name="last\_name" id="last\_name">**



Next, let's add a field for taking in emails. First, we add a label.

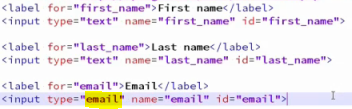
**Add the following bolded tags below the last input tag:**

**<label for="email">Email</label>**

And next we add in the field itself.

**Add the following bolded tags below the last label tag:**

**<input type="email" name="email" id="email">**



Notice this time we're not using the `text` value for the type; we're using the `email` type. Using this type allows mobile devices to display the correct keyboard. The symbols that are most frequently used in entering emails will be made available on the keyboard. It also lets us do some rudimentary email validation.

Some browsers don't make use of this field, but they just treat them as regular text boxes. Everything still works. Use this whenever you take in an email address.

**Save the contact.html page and refresh the browser**.



Next, let's make a dropdown list that lets people specify the reason they are contacting us.

To do this we have to use a couple of tags. The `select` tag defines the element.

So, let's add a label and the select tag.

**Add the following bolded tags below the last input tag:**

**<label for="reason">Reason</label>**

**<select id="reason" name="reason">**

**</select>**

I just want to point out here that the label is a sibling element. The label doesn't contain the form field, so we don't indent the form field farther than the label. However, labels can contain form fields instead of using the `for` attribute.

It's legal to do the form field this way:

<label for="reason">Reason

<select id="reason" name="reason">

</select>

</label>

But we won't be doing that in this class. It makes styling the form very difficult if you do that.

So, this gives us the field but now we have to specify the various options.

Inside the `select` tags we use the `option` tag.

<option value="inquiry">General Question</option>

<option value="complaint">Complaint</option>

<option value="jobs">Job Inquiry</option>

Notice that the options have values. These values are what gets sent to the server-side script. The text the user sees is between the opening and closing option tag. The values might be specific to the server-side script. Notice in our example

we have the text being "General Question" but the value that gets sent is called "inquiry". This lets us change what the user sees while not having to change the server-side part

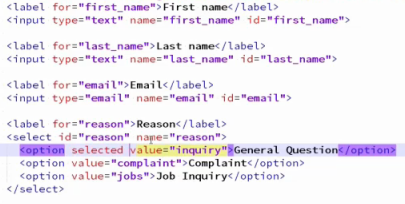
We can also set one to be selected by default.

**Add the following bolded inside the select tag:**

**<option selected value="inquiry">General Question</option>**

**<option value="complaint">Complaint</option>**

**<option value="jobs">Job Inquiry</option>**



**Save the contact.html page and refresh the browser**.



Let's look at our form in the browser. It's looking pretty good. Here are our fields and here are our dropdown options. To add more, we just add more options to the markup.

If you wanted a dropdown of all 50 states, you'd just go make one the same way, by coding up 50 options. In reality, we don't do that. Web forms are often something you build in a server-side language where you combine HTML with code, and you'll write a little program that will pull the states from some database or other collection of data and generate that part of the page for you. You'll learn how to do that in a future class.

Next, let's ask if the person filling out this form already works here. We can use this information to route the request to their manager or something.

We’ll probably just fire them for complaining.

**Add the following bolded tags below the last select tag:**

**<p>Are you already employed here? </p>**

**<input type="radio"id="employed\_yes" name="employed" value="yes">**

**<label class="inline" for="employed\_yes">Yes</label>**

**<input type="radio"id="employed\_no" name="employed" value="no">**

**<label class="inline" for="employed\_no">No</label>**

Radio buttons are named after the old preset buttons on car radios, where only one could be pressed at a time. Pressing one button depresses all of the others. We give each radio button the same name attribute, which creates the group of radio buttons. When the form is submitted, the browser only sends the value of the radio button that was active.

Now here's the really important thing about labels - they're great for accessibility for screen readers, but they're also great for accessibility for everyone else. When a label is associated with a radio button, we can click the label to activate the radio button.



Now let's move on to checkboxes. In order to send this data to us, we're going to make the end user agree to our terms of service, which says something like "We get to own all of your ideas you send us and we don't have to pay you." Or something equally nefarious.

To make a checkbox, we're going to use an `input` tag again, but this time we'll use the checkbox type. I'm going to put the label after the checkbox this time.

**Add the following bolded tags below the last label tag:**

**<input type="checkbox" name="terms" id="terms" value="accepted">**

**<label class="inline" for="terms">**

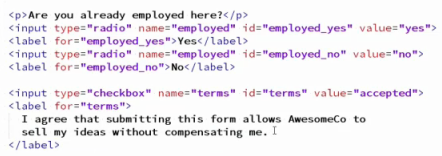
**I agree that submitting this form allows AwesomeCo**

**to sell my ideas without compensating me.**

**</label>**

Just like radio buttons, we can select the label and it will toggle the checkbox, which is really nice.

If we wanted to have multiple checkboxes on a page, we could code them just like radio buttons and give each button its own ID and value, but use the same name attribute. Then the browser would send the values of all checked boxes to the server as a collection.



**Save the contact.html page and refresh the browser**.



Next, we need the actual comment field so we can collect that all-important data from our visitor. We do this with the `textarea` element.

**Add the following bolded tags below the last label tag:**

**<label for="the\_message">Message</label>**

**<textarea id="the\_message" name="the\_message"></textarea>**

The textarea is a little different, as you can see. It has a closing tag, and the value is the content between the opening and closing tags!

When we look at this in the browser we see the field for people to enter in the data. We could specify rows and columns of text in the HTML, but it's better to style this form with CSS.

This is a good time to remind you that the `name` values we used in these form fields are required by the specific server-side script I'm sending this data to. A different server-side script would require different values for the form field name attributes.

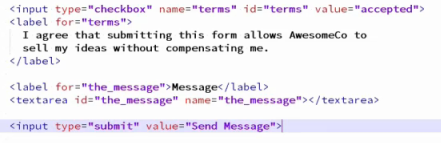
Also, the form field ID doesn't have to match the name attribute. That's just a convention I use.

Now, in order to submit this form to the server, we need a button. And we can do that by adding another `input` tag:

**Add the following bolded tags below the last textarea tag:**

**<input type="submit" value="Send Message">**

The value becomes the text we see on the button. We don't need a `name` attribute here because we don't need to use this button to send any data. We also don't need an ID here because we don't have to associate this with a label.

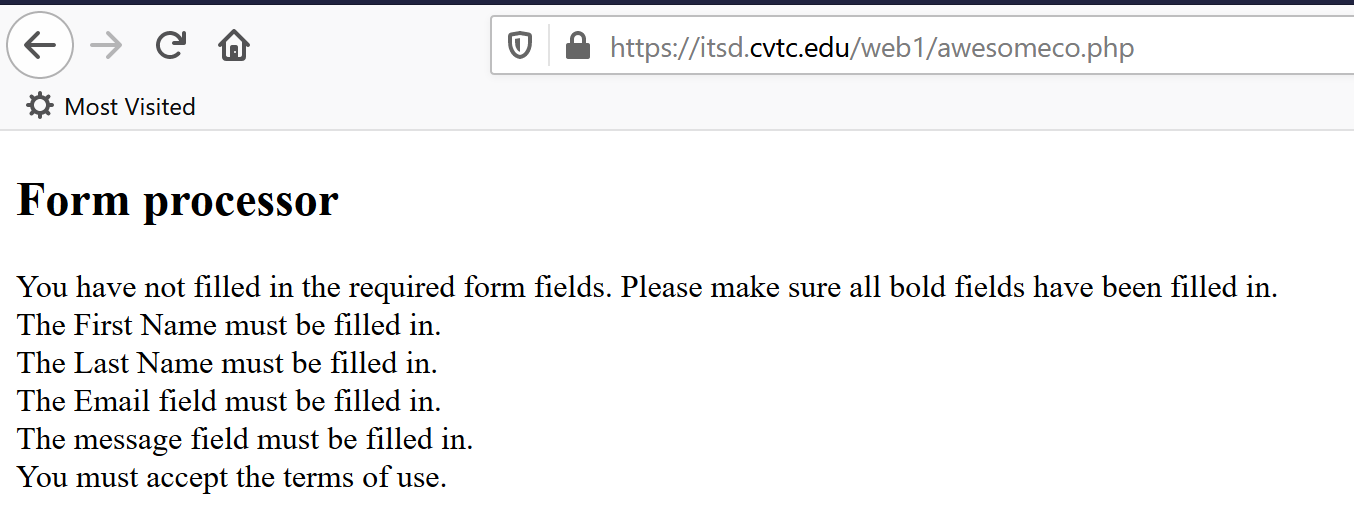


So, there's our simple form.

**Save the contact.html page and refresh the browser**.



Let's see what happens when we submit, **press the button**, the empty form to the server:



First thing you might notice is that the URL changed. By submitting the form we've essentially followed a link to a new page. The browser took our form information and made a request to this new page.

The script on the server tells us that we forgot to put in some information. The server-side script is validating the input it got to make sure the input was good. But now we have to press the back button and remember what we need to enter. This is kind of a cruddy user interface. We need to tell our visitor exactly what the required information is so they

can get it right the first time.

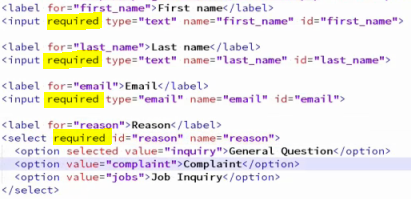
In order for us to get back to where we came from, we need to use our browser's "back" button. If you had control over both sides of the system, you would develop it so that it redisplays the form right here so your visitor could fill it in again. But we're not to that point yet, so we'll have to work with what we have.

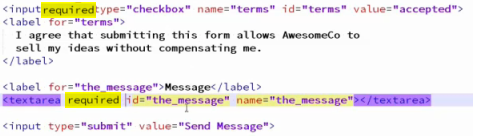
Let's tweak things a bit to make the user interface better.

**Required Fields**

We can make fields required by adding the `required` attribute to a form field. The `required` attribute doesn't have a value. We just add it on to any form field we want to be a required field. This only works in modern browsers though.

**Add required attribute to the following input, select, and textarea tags: (first\_name, last\_name, email, reason, terms, the\_message).**

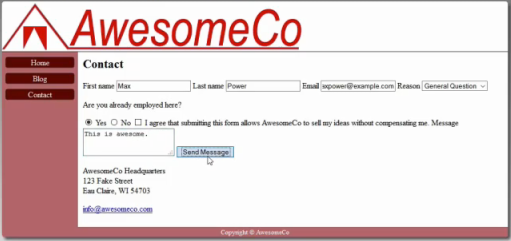




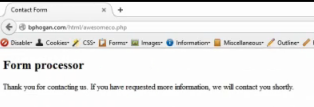
**Save the contact.html page and refresh the browser**.



Now when we try to submit the form it doesn't let us. We have to enter the information required.

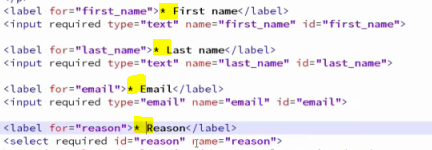


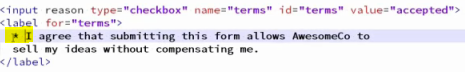
When all the required fields are populated and the button is pressed, the following screen should show:



So, let's add asterisks to each label of each required field.

**Add an \* inside the labels as shown below:**





Then we'll put a paragraph at the top saying that all required fields have an asterisk.

**Add the following bolded tag below above the form tag:**

**<p>All fields with an asterisk (\*) are required. </p>**

****

Now it's really important to remember that we have to check the validity of user information on both the client and server side. These required attributes are only available on some browsers, and by modifying the HTML in the web console, a knowledgeable user can remove these required attributes from the page.

They can't, however, touch our server-side code without hacking our server. So, we're only adding this client-side validation to improve the interface.

**Placeholders**

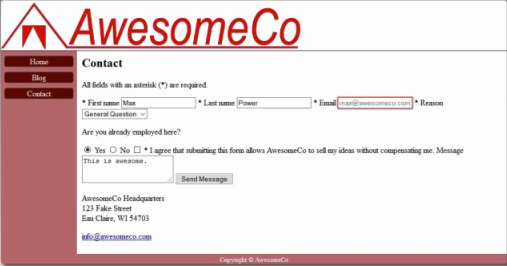
Let's add some hints to the fields. By using the placeholder attribute, we can instruct the user how they should fill in the information.

**Add placeholder attribute shown in bold to input tag – id=”email”.**

<input type="email" name="email" id="email" **placeholder="max@awesomeco.com"**>



**Save contacts.html and refresh the browser**.



We now see the placeholder text when the field is empty.

**Styling**

Let’s style the form so that each field sits below its label, except for radio buttons and checkboxes. Leave those alone.

Labels and fields are all are inline elements. But if we wanted them to be on their own line, we could use CSS.

**Open styles.css in your text editor.**

**Add the following selector to the bottom of the CSS document.**

**label {**

**display: block;**

**}**

Next, we can use CSS attribute selectors to identify specific CSS elements.

**Add the following selector to the bottom of the CSS document.**

**input[type=radio] + label {**

**display: inline;**

**}**

We can find all the input fields where the type is radio and find its adjacent label using the plus (+) operator. Since we've set all labels to be block, we can switch this back to inline.

And since the checkbox works the same way, we can just add another selector to the rule.

**Add the following selector shown in bold to the already existing input[type=radio] selector.**

input[type=radio] + label, **input[type=checkbox] + label** {

display: inline;

}

When we save and refresh in the browser, we see that we're almost where we want to be. However, the submit button should really be on its own line, and so should the checkbox.

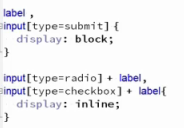
The submit button is easy.... we can use the same attribute selector trick to make the submit button display as a block element. We can just add the selector for that to the existing rule we have for labels.

**Add the following input[type=submit] selector shown in bold to the already defined label selector.**

label, **input[type=submit] {**

display: block;

}



However, to get terms of service box on its own line, there's very little we can do in the CSS. Form fields are largely controlled by the browser.

Since we've reached the limit of what we can do with CSS, we'll have to go back to the markup and make an adjustment.

**Save the style.css page.**

**Better Organization**

There are two other HTML elements we can use for organizing our forms, the `fieldset` and the `legend`. The fieldset lets us group a section of a form in a logical manner, and the `legend` lets us label that section.

If we wrap our personal information in a fieldset, and then wrap the terms of service in a fieldset, we'll have two sections of the form we can easily style.

So, let's find the start of the form.

**Go back to the contact.html page.**

**Add the opening <fieldset> above the first label, first\_name.**

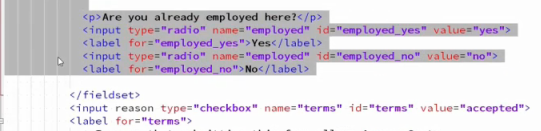
**<fieldset>**



And place the closing fieldset after the end of the personal information.

**Add the closing </fieldset> between employed and terms fields.**

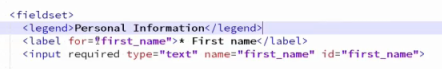
**</fieldset>**



**Indent all the children tags inside fieldset two spaces by highlighting those rows and pressing the tab key.**

Then we can add a legend element to this, identifying it as the personal info region.

**Add the following legend tag under the open fieldset.**



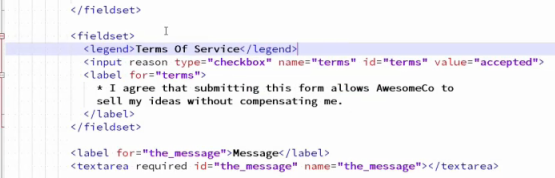
Then we can wrap the terms of service in its own fieldset. And we can add a legend to this to label this as the terms of service.

**Add a fieldset around terms of Service under the above </fieldset>.**

**Indent tags inside fieldset two spaces.**

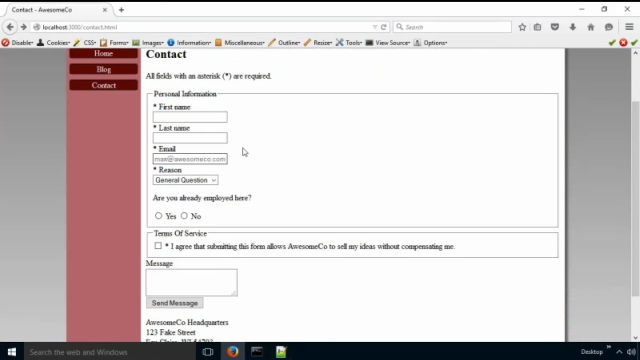
**Add a legend to the new fieldset.**

**<legend>Terms of Service</legend>**



And now we have a couple new elements we can hook styles onto.

**Save contact.html and refresh the browser.**



The browser has its own way of styling these fieldsets, and we have some control over these in CSS.

We can remove the border from the fieldset, and alter its default padding. We'll zero out all the padding except for the bottom, which we'll set to 1%.

**Add the following selector to the bottom of your style.css document.**

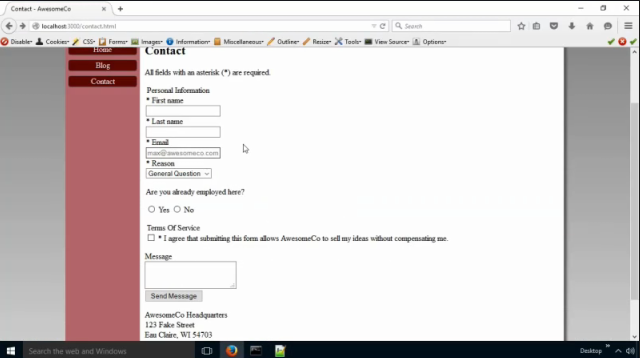
**fieldset {**

**border: none;**

**padding: 0 0 1em 0;**

**}**

**Save style.css and refresh in the browser and things look great.**



Once again, thanks to properly marked-up HTML, the CSS is simple. No need to inject break tags into the page.

And that's it for forms. Remember, a form needs to work in conjunction with some server-side processing script. You'll learn to write those server-side pieces in a future class. For now, you'll just work with server-side scripts provided for you and you'll have to tailor your HTML code to work with those scripts.

Below is the finished contact.html and style.css documents.

**contact.html page:**

<!DOCTYPE html>

<html lang="en-US">

<head>

<meta charset="utf-8">

<meta name="viewport"

content="width=device-width, initial-scale=1.0">

<title>Contact - AwesomeCo</title>

<link rel="stylesheet" href="stylesheets/style.css">

</head>

<body>

<div id="wrapper">

<header>

<h1>AwesomeCo</h1>

</header>

<nav>

<ul>

<li><a href="index.html">Home</a></li>

<li><a href="blog.html">Blog</a></li>

<li><a href="contact.html">Contact</a></li>

</ul>

</nav>

<main>

<h2>Contact</h2>

**<p>All fields with an asterisk (\*) are required.</p>**

**<form method="post" action="https://itsd.cvtc.edu/web1/awesomeco.php ">**

**<fieldset>**

**<legend>About you</legend>**

**<label for="first\_name">\* First Name</label>**

**<input placeholder="Max" required id="first\_name" type="text" name="first\_name">**

**<label for="last\_name">\* Last Name</label>**

**<input placeholder="Power" required id="last\_name" type="text" name="last\_name">**

**<label for="email">\* Email</label>**

**<input placeholder="max@awesomeco.com" required id="email" type="email" name="email">**

**<label for="reason">Reason</label>**

**<select id="reason" name="reason">**

**<option selected value="inquiry">General Question</option>**

**<option value="complaint">Complaint</option>**

**<option value="jobs">Job inquiry</option>**

**</select>**

**<p>Are you already employed here?</p>**

**<input type="radio" id="employed\_yes" name="employed" value="yes">**

**<label for="employed\_yes">Yes</label>**

**<input type="radio" id="employed\_no" name="employed" value="no">**

**<label for="employed\_no">No</label>**

**</fieldset>**

**<fieldset>**

**<legend>Terms of service</legend>**

**<input required type="checkbox" id="terms" name="terms" value="yes">**

**<label for="terms">**

**\* I agree that submitting this form allows AwesomeCo to**

**sell my ideas without compensating me.**

**</label>**

**</fieldset>**

**<label for="the\_message">\* Message</label>**

**<textarea required id="the\_message" name="the\_message"></textarea>**

**<input type="submit" value="Send Message">**

**</form>**

<section>

<p>

AwesomeCo Headquarters<br>

123 Fake Street<br>

Eau Claire WI, 54703

</p>

<p>

Send email to us at

<a href="mailto:info@awesomeco.com">info@awesomeco.com</a>

</p>

</section>

</main>

<footer>

<small>Copyright &copy; AwesomeCo</small>

</footer>

</div>

</body>

</html>

**style.css:**

h1, h2, h3, h4 {

margin-top: 0;

}

#wrapper > header {

background-color: #FFF;

background-image: url( ../images/awesomeco\_logo\_mobile.png);

background-repeat: no-repeat;

background-position: center;

background-size: contain;

border-bottom: 1px solid #300;

height: 70px;

}

header > h1 {

text-indent: -9999px;

}

nav > ul {

list-style: none;

margin: 0;

padding: 0;

}

nav > ul > li {

border: 1px solid #333;

border-radius: 5px;

margin: 10px auto;

text-align: center;

width: 90%;

}

nav > ul > li > a {

background-color: #600;

color: #FFF;

display: block;

text-decoration: none;

}

#wrapper {

background-color: #BC7277;

}

main {

background-color: #FFF;

border-radius: 5px;

margin: 0 auto;

padding: 1%;

width: 88%;

}

footer {

color: #FFF;

text-align: center;

}

main > img {

max-width: 100%;

height: auto;

}

@media only screen and (max-width: 767px) {

h1, h2, h3, h4 {

text-align: center;

}

}

/\* For tablets and up \*/

@media only screen and (min-width: 768px) {

#wrapper > header {

background-image: url(../images/awesomeco\_logo.png);

background-size: auto;

height: 100px;

}

main img {

float: left;

margin-right: 1%;

}

main ul {

overflow: hidden;

}

}

/\* tablets only \*/

@media only screen and (min-width: 768px) and (max-width: 959px) {

nav > ul > li {

display: inline-block;

width: 20%;

}

nav {

text-align: center;

}

}

/\* desktop sizes \*/

@media only screen and (min-width: 960px) {

body {

background-color: #DDD;

background-image: linear-gradient(to bottom, #888, #DDD);

background-repeat: no-repeat;

}

#wrapper {

border-radius: 5px;

box-shadow: 5px 5px 5px #555;

margin: 0 auto;

overflow: hidden;

width: 80%;

}

#wrapper > header {

background-position: left;

}

nav {

float: left;

width: 15%;

}

main {

margin-left: 15%;

border-radius: 0;

width: 83%;

}

}

**label, input[type=submit] {**

**display: block;**

**}**

**input[type=radio] + label , input[type=checkbox] + label {**

**display: inline;**

**}**

**fieldset {**

**border: none;**

**padding: 0 0 1em 0;**

**}**