

HTML FORMS

Nº 13. OF HTML & CSS IS HARD

A friendly web development tutorial for capturing user input

HTML form elements let you collect input from your website's visitors. Mailing lists, contact forms, and blog post comments are common examples for small websites, but in organizations that rely on their website for revenue, forms are sacred and revered.

TEXT INPUT

RADIO BUTTONS

- ☐ Option One
☒ Option Two

DROPDOWN MENU

TEXTAREA

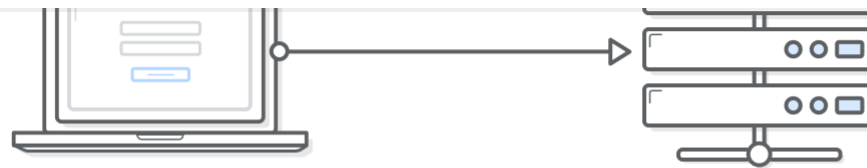
Lots of text input. Magnis sit ultricies scelerisque vitae consectetur montes taciti elit. A sapien in suspendisse mauris sem posuere dapibus.

CHECKBOXES

- ☒ Option One
☐ Option Two
☒ Option Three

BUTTON

Forms are the “money pages.” They’re how e-commerce sites sell their products, how SaaS companies collect payment for their service, and how non-profit groups raise money online. Many companies measure the success of their website by the effectiveness of its forms because they answer questions like “how many leads did our website send to our sales team?” and “how many people signed up for our product last week?” This often means that forms are subjected to endless A/B tests and optimizations.



FORM ELEMENTS

(FRONTEND HTML & CSS)

FORM PROCESSING

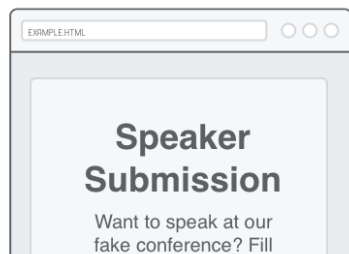
(BACKEND SERVER)

There are two aspects of a functional HTML form: the frontend user interface and the backend server. The former is the *appearance* of the form (as defined by HTML and CSS), while the latter is the code that processes it (storing data in a database, sending an email, etc). We'll be focusing entirely on the frontend this chapter, leaving backend form processing for a future tutorial.

SETUP

Unfortunately, there's really no getting around that fact that styling forms is *hard*. It's always a good idea to have a mockup representing the exact page you want to build before you start coding it up, but this is particularly true for forms. So, here's the example we'll be creating in this chapter:

MOBILE/TABLET

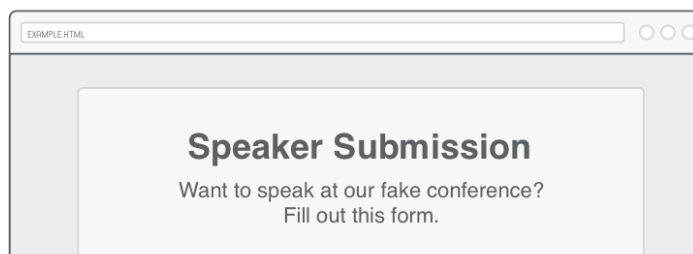


EXAMPLE.HTML

Speaker Submission

Want to speak at our fake conference? Fill

DESKTOP



EXAMPLE.HTML

Speaker Submission

Want to speak at our fake conference?
Fill out this form.



The image displays two versions of a speaker submission form side-by-side. The left version is a wireframe, showing the layout with labels and empty input fields. The right version is a styled version, featuring a light blue background, rounded corners, and pre-filled data for demonstration.

Form Fields:

- Name:** A text input field.
- Email:** A text input field containing "you@example.com".
- Type of Talk:** Two radio buttons labeled "Main Stage" and "Workshop".
- T-Shirt Size:** A dropdown menu with "Extra Small" selected.
- Abstract:** A large text area for describing the talk.
- Submit:** A blue button labeled "Submit".

Additional Text:

- Below the abstract text area: "Describe your talk in 500 words or less".
- Below the abstract text area: A checkbox labeled "I'm actually available the date of the talk".

As you can see, this is a speaker submission form for a fake conference. It hosts a pretty good selection of HTML forms elements: various types of text fields, a group of radio buttons, a dropdown menu, a checkbox, and a submit button.



speaker-submission.html. For starters, let's add the markup for the header.
(Hey look! It has [some semantic HTML!](#))

```
<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='UTF-8' />
    <title>Speaker Submission</title>
    <link rel='stylesheet' href='styles.css' />
  </head>
  <body>
    <header class='speaker-form-header'>
      <h1>Speaker Submission</h1>
      <p><em>Want to speak at our fake conference? Fill out
        this form.</em></p>
    </header>
  </body>
</html>
```

Next, create a `styles.css` file and add the following CSS. It uses a simple [flexbox](#) technique to center the header (and form) no matter how wide the browser window is:

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```



```
color: #5D6063;
background-color: #EAEDF0;
font-family: "Helvetica", "Arial", sans-serif;
font-size: 16px;
line-height: 1.3;

display: flex;
flex-direction: column;
align-items: center;
}

.speaker-form-header {
  text-align: center;
  background-color: #F6F7F8;
  border: 1px solid #D6D9DC;
  border-radius: 3px;

  width: 80%;
  margin: 40px 0;
  padding: 50px;
}

.speaker-form-header h1 {
  font-size: 30px;
  margin-bottom: 20px;
}
```





discussed in the *Responsive Design* chapter. These base CSS rules give us our mobile layout and provide a foundation for the desktop layout, too. We'll create the media query for a fixed-width desktop layout later in the chapter.

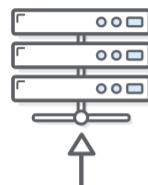
HTML FORMS

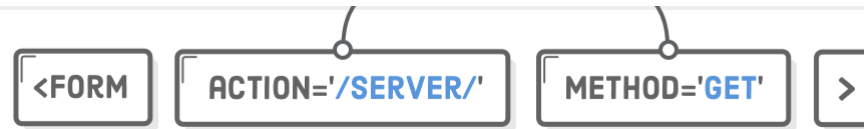
On to forms! Every HTML form begins with the aptly named `<form>` element. It accepts a **number of attributes**, but the most important ones are **action** and **method**. Go ahead and add an empty form to our HTML document, right under the `<header>`:

```
<form action='' method='get' class='speaker-form'>
</form>
```

The **action** attribute defines the URL that processes the form. It's where the input collected by the form is sent when the user clicks the **Submit** button. This is typically a special URL defined by your web server that knows how to process the data. Common backend technologies for processing forms include **Node.js**, **PHP**, and **Ruby on Rails**, but again, we'll be focusing on the frontend in this chapter.

BACKEND SERVER





The `method` attribute can be either `post` or `get`, both of which define how the form is submitted to the backend server. This is largely dependent on how your web server wants to handle the form, but the general rule of thumb is to use `post` when you're *changing* data on the server, reserving `get` for when you're only *getting* data.

By leaving the `action` attribute blank, we're telling the form to submit to the same URL. Combined with the `get` method, this will let us inspect the contents of the form.

STYLING FORMS

Of course, we're looking at an empty form right now, but that doesn't mean we can't add some styles to it like we would a container `<div>`. This will turn it into a box that matches our `<header>` element:

```
.speaker-form {  
  background-color: #F6F7F8;  
  border: 1px solid #D6D9DC;  
  border-radius: 3px;  
  
  width: 80%;  
  padding: 50px;
```



TEXT INPUT FIELDS

To actually collect user input, we need a new tool: the `<input/>` element. Insert the following into our `<form>` to create a text field:

```
<div class='form-row'>
  <label for='full-name'>Name</label>
  <input id='full-name' name='full-name' type='text' />
</div>
```

First, we have a container `<div>` to help with styling. This is pretty common for separating input elements. Second, we have a `<label>`, which you can think of as another **semantic HTML element**, like `<article>` or `<figcaption>`, but for form labels. A label's `for` attribute must match the `id` attribute of its associated `<input/>` element.





elements we've encountered because it can dramatically change appearance depending on its `type` attribute, but it always creates some kind of interactive user input. We'll see other values besides `text` throughout the chapter. Remember that **ID selectors are bad**—the `id` attribute here is *only* for connecting it to a `<label>` element.



Conceptually, an `<input/>` element represents a “variable” that gets sent to the backend server. The `name` attribute defines the name of this variable, and the value is whatever the user entered into the text field. Note that you can pre-populate this value by adding a `value` attribute to an `<input/>` element.

STYLING TEXT INPUT FIELDS

An `<input/>` element can be styled like any other HTML element. Let's add some CSS to `styles.css` to pretty it up a bit. This makes use of all the concepts from the [Hello, CSS](#), [Box Model](#), [CSS Selectors](#), and [Flexbox](#) chapters:

```
.form-row {  
  margin-bottom: 40px;
```



```
flex-direction: column;
flex-wrap: wrap;
}

.form-row input[type='text'] {
  background-color: #FFFFFF;
  border: 1px solid #D6D9DC;
  border-radius: 3px;
  width: 100%;
  padding: 7px;
  font-size: 14px;
}

.form-row label {
  margin-bottom: 15px;
}
```

The `input[type='text']` part is a new type of CSS selector called an “attribute selector”. It only matches `<input/>` elements that have a `type` attribute equal to `text`. This lets us specifically target text fields opposed to radio buttons, which are defined by the same HTML element (`<input type='radio' />`). You can read more about attribute selectors at [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics/attribute_selectors).

All of our styles are “namespaced” in a `.form-row` descendant selector. Isolating `<input/>` and `<label>` styles like this makes it easier to create



`input[type="text"]` and `label` selectors once we get to `radio` buttons.

Finally, let's tweak these base styles to create our desktop layout. Add the following [media query](#) to the end of our stylesheet.

```
@media only screen and (min-width: 700px) {  
  .speaker-form-header,  
  .speaker-form {  
    width: 600px;  
  }  
  .form-row {  
    flex-direction: row;  
    align-items: flex-start; /* To avoid stretching */  
    margin-bottom: 20px;  
  }  
  .form-row input[type='text'] {  
    width: 250px;  
    height: initial;  
  }  
  .form-row label {  
    text-align: right;  
    width: 120px;  
    margin-top: 7px;  
    padding-right: 20px;  
  }  
}
```



`<label>` appear on top of its `<input/>` element in the mobile layout, but to the left of it in the desktop layout.

EMAIL INPUT FIELDS

The `<input/>` element's `type` attribute also lets you do basic input validation. For example, let's try adding another input element that *only* accepts email addresses instead of arbitrary text values:

```
<div class='form-row'>
  <label for='email'>Email</label>
  <input id='email'
    name='email'
    type='email'
```



This works exactly like the `type='text'` input, except it automatically checks that user entered an email address. In Firefox, you can try typing something that's not an email address, then clicking outside of the field to make it lose focus and validate its input. It should turn red to show the user that it's an incorrect value. Chrome and Safari don't attempt to validate until user tries to submit the form, so we'll see this in action later in this chapter.

A screenshot of a web form with two input fields. The first field is labeled 'Name' and is empty. The second field is labeled 'Email' and contains the text 'notanemail'. The 'Email' field has a red border around it, indicating a validation error. The form is set against a light gray background with rounded corners.

This is more than just validation though. By telling browsers that we're looking for an email address, they can provide a more intuitive user experience. For instance, when a smartphone browser sees this `type='email'` attribute, it gives the user a special email-specific keyboard with an easily-accessible @ character.

Also notice the new `placeholder` attribute that lets you display some default text when the `<input/>` element is empty. This is a nice little UX technique to prompt the user to input their own value.

which you can read about on MDN's [<input/> reference](#). Of particular interest are the `required`, `minlength`, `maxlength`, and `pattern` attributes.

STYLING EMAIL INPUT FIELDS

We want our email field to match our text field from the previous section, so let's add another attribute selector to the existing `input[type='text']` rule, like so:

```
/* Change this rule */
.form-row input[type='text'] {
  background-color: #FFFFFF;
  /* ... */
}

/* To have another selector */
.form-row input[type='text'],
.form-row input[type='email'] {
  background-color: #FFFFFF;
  /* ... */
}
```

Again, we don't want to use a plain old `input` type selector here because that would style *all* of our `<input/>` elements, including our upcoming radio buttons and checkbox. This is part of what makes styling forms tricky. Understanding the CSS to pluck out exactly the elements you want is a crucial skill.



`input[type='text']` rule in our media query to match the following (note that we're preparing for the next few sections with the `select`, and `textarea` selectors):

```
@media only screen and (min-width: 700px) {  
  /* ... */  
  .form-row input[type='text'],  
  .form-row input[type='email'],    /* Add */  
  .form-row select,                /* These */  
  .form-row textarea {             /* Selectors */  
    width: 250px;  
    height: initial;  
  }  
  /* ... */  
}
```

Since we can now have a “right” and a “wrong” input value, we should probably convey that to users. The `:invalid` and `:valid` [pseudo-classes](#) let us style these states independently. For example, maybe we want to render both the border and the text with a custom shade of red when the user entered an unacceptable value. Add the following rule to our stylesheet, outside of the media query:

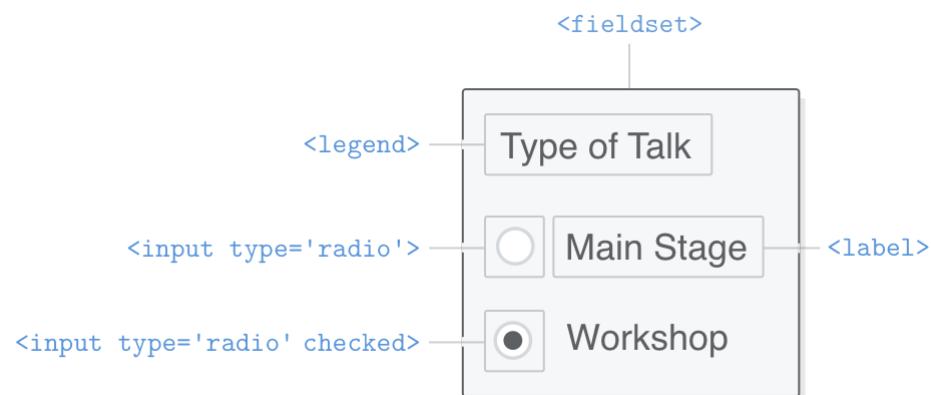
```
.form-row input[type='text']:invalid,  
.form-row input[type='email']:invalid {  
  border: 1px solid #D55C5F;  
  color: #D55C5F;
```



Until we include a submit button, you'll only be able to see this in Firefox, but you get the idea. There's a similar pseudo-class called `:focus` that selects the element the user is currently filling out. This gives you a lot of control over the appearance of your forms.

RADIO BUTTONS

Changing the `type` property of the `<input/>` element to `radio` transforms it into a radio button. Radio buttons are a little more complex to work with than text fields because they always operate in groups, allowing the user to choose one out of many predefined options.



This means that we not only need a label for each `<input/>` element, but also a way to group radio buttons and label the entire group. This is what the



create should.

- Be wrapped in a `<fieldset>`, which is labeled with a `<legend>`.
- Associate a `<label>` element with each radio button.
- Use the same `name` attribute for each radio button in the group.
- Use different `value` attributes for each radio button.

Our radio button example has all of these components. Add the following to our `<form>` element underneath the email field:

```
<fieldset class='legacy-form-row'>
  <legend>Type of Talk</legend>
  <input id='talk-type-1'
        name='talk-type'
        type='radio'
        value='main-stage' />
  <label for='talk-type-1' class='radio-label'>Main Stage</label>
  <input id='talk-type-2'
        name='talk-type'
        type='radio'
        value='workshop'
        checked />
  <label for='talk-type-2' class='radio-label'>Workshop</label>
</fieldset>
```

Unlike text fields, the user can't enter custom values into a radio button, which is why each one of them needs an explicit `value` attribute. This is the value that



important that each radio button has the same `name` attribute, otherwise the form wouldn't know they were part of the same group.

We also introduced a new attribute called `checked`. This is a “boolean attribute”, meaning that it never takes a value—it either exists or doesn't exist on an `<input/>` element. If it does exist on either a radio button or a checkbox element, that element will be selected/checked by default.

STYLING RADIO BUTTONS

We have a few things working against us with when it comes to styling radio buttons. First, there's simply more elements to worry about. Second, the `<fieldset>` and `<legend>` elements have rather ugly default styles, and there's not a whole lot of consistency in these defaults across browsers. Third, at the time of this writing, `<fieldset>` doesn't support flexbox.

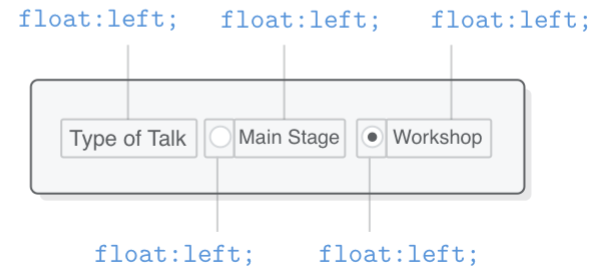
But don't fret! This is a good example of `floats` being a useful fallback for legacy/troublesome elements. You'll notice that we didn't wrap the radio buttons in our existing `.form-row` class, opting instead for a new `.legacy-form-row` class. This is because it's going to be completely separate from our other elements, using floats instead of flexbox.

.E LAYOUT





DESKTOP LAYOUT



Start with the mobile and tablet styles by adding the following rules outside of our media query. We want to get rid of the default `<fieldset>` and `<legend>` styles, then float the radio buttons and labels so they appear in one line underneath the `<legend>`:

```
.legacy-form-row {  
  border: none;  
  margin-bottom: 40px;  
}  
  
.legacy-form-row legend {  
  margin-bottom: 15px;  
}  
  
.legacy-form-row .radio-label {  
  display: block;  
  font-size: 14px;  
}
```



```
.legacy-form-row input[type='radio'] {  
  margin-top: 2px;  
}  
  
.legacy-form-row .radio-label,  
.legacy-form-row input[type='radio'] {  
  float: left;  
}
```

For the desktop layout, we need to make the `<legend>` line up with the `<label>` elements in the previous section (hence the `width: 120px` line), and we need to float *everything* to the left so they appear on the same line. Update our media query to include the following:

```
@media only screen and (min-width: 700px) {  
  /* ... */  
  .legacy-form-row {  
    margin-bottom: 10px;  
  }  
  .legacy-form-row legend {  
    width: 120px;  
    text-align: right;  
    padding-right: 20px;  
  }  
  .legacy-form-row legend {  
    float: left;  
  }
```



As far as layouts go, this is a pretty good cross-browser solution. However, customizing the appearance of the actual button is another story. It's possible by taking advantage of the `checked` attribute, but it's a little bit complicated. We'll leave you to Google "custom radio button CSS" and explore that rabbit hole on your own.

SELECT ELEMENTS (DROPDOWN MENUS)

Dropdown menus offer an alternative to radio buttons, as they let the user select one out of many options. The `<select>` element represents the dropdown menu, and it contains a bunch of `<option>` elements that represent each item.

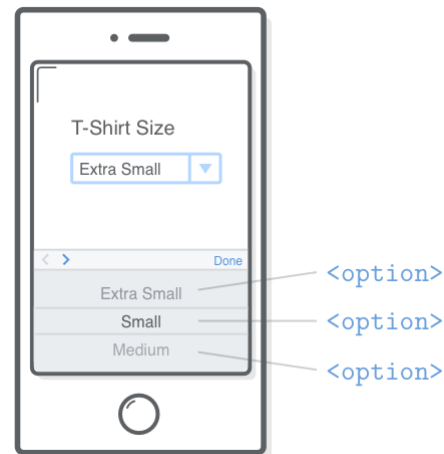
```
<div class='form-row'>
  <label for='t-shirt'>T-Shirt Size</label>
  <select id='t-shirt' name='t-shirt'>
    <option value='xs'>Extra Small</option>
    <option value='s'>Small</option>
    <option value='m'>Medium</option>
    <option value='l'>Large</option>
  </select>
</div>
```

Just like our radio button `<input/>` elements, we have `name` and `value` attributes that get passed to the backend server. But, instead of being defined on



STYLING SELECT ELEMENTS

And, also just like our radio buttons, `<select>` elements are notoriously hard to style. However, there's a reason for this. Dropdowns are a complex piece of interactivity, and their behavior changes significantly across devices. For instance, on an iPhone, clicking a `<select>` element brings up a native scrolling UI component that makes it much easier to navigate the menu.



It's usually a good idea to let the browser/device determine the best way to preset a `<select>` element, so we'll be keeping our CSS pretty simple. Unfortunately, even the simplest things are surprisingly hard. For instance, try changing the font size of our `<select>` element:

```
.form-row select {  
  width: 100%;
```



```
}
```

This will work in Firefox, but not in Chrome or Safari! To sort of fix this, we can use a vendor-specific prefix for the `appearance` property:

```
.form-row select {  
  width: 100%;  
  padding: 5px;  
  font-size: 14px;           /* This won't work in Chrome or Safari */  
  -webkit-appearance: none; /* This will make it work */  
}
```

The `-webkit` prefix will *only* apply to Chrome and Safari (which are powered by the WebKit rendering engine), while Firefox will remain unaffected. This is effectively a hack, and even MDN says [not to use this CSS property](#).

Style difficulties like this are a serious consideration when building a form. If you need custom styles, you may be better off using radio buttons or JavaScript UI widgets. [Bootstrap Dropdowns](#) and [jQuery Selectmenu's](#) are common JavaScript solutions for customizing select menus. In any case, at least you now understand the problem. You can read more about `<select>` issues [here](#).

TEXTAREAS



amounts of text from the user. They're suitable for things like biographies, essays, and comments. Go ahead and add a `<textarea>` to our form, along with a little piece of instructional text:

```
<div class='form-row'>
  <label for='abstract'>Abstract</label>
  <textarea id='abstract' name='abstract'></textarea>
  <div class='instructions'>Describe your talk in 500 words or less</div>
</div>
```

Note that this isn't self-closing like the `<input/>` element, so you always need a closing `</textarea>` tag. If you want to add any default text, it needs to go *inside* the tags opposed to a `value` attribute.

STYLING TEXTAREAS

Fortunately, styling textareas is pretty straightforward. Add the following to our `styles.css` file (before the media query):

```
.form-row textarea {
  font-family: "Helvetica", "Arial", sans-serif;
  font-size: 14px;

  border: 1px solid #D6D9DC;
  border-radius: 3px;

  min-height: 200px;
```




```
    resize: none;
  }

  .form-row .instructions {
    color: #999999;
    font-size: 14px;
    margin-bottom: 30px;
  }
```

By default, many browsers let the user resize `<textarea>` elements to whatever dimensions they want. We disabled this here with the `resize` property.

We also need a little tweak in our desktop layout. The `.instructions <div>` needs to be underneath the `<textarea>`, so let's nudge it left by the width of the `<label>` column. Add the following rule to the end of our media query:

```
@media only screen and (min-width: 700px) {
  /* ... */
  .form-row .instructions {
    margin-left: 120px;
  }
}
```

CHECKBOXES



option, they let the user pick as many as they want. This simplifies things, since the browser doesn't need to know which checkboxes are part of the same group. In other words, we don't need a `<fieldset>` wrapper or shared `name` attributes. Add the following to the end of our form:

```
<div class='form-row'>
  <label class='checkbox-label' for='available'>
    <input id='available'
      name='available'
      type='checkbox'
      value='is-available' />
    <span>I'm actually available the date of the talk</span>
  </label>
</div>
```

The way we used `<label>` here was a little different than previous sections. Instead of being a separate element, the `<label>` wraps its corresponding `<input/>` element. This is perfectly legal, and it'll make it easier to match our desired layout. It's still a best practice to use the `for` attribute.

STYLING CHECKBOXES

For the mobile layout, all we need to do is override the `margin-bottom` that we put on the rest the `<label>` elements. Add the following to `styles.css`, outside of the media query:



```
}
```

And inside the media query, we have to take that 120-pixel label column into account:

```
@media only screen and (min-width: 700px) {  
  /* ... */  
  .form-row .checkbox-label {  
    margin-left: 120px;  
    width: auto;  
  }  
}
```

By wrapping both the checkbox and the label text, we're able to use a `width: auto` to make the entire form field be on a single line (remember that the `auto` width makes the box match the size of its contents).

EXAMPLE.HTML

Speaker Submission

Want to speak at our fake conference?
Fill out this form.

Name



Type of Talk ☐ Main Stage ☐ Workshop

T-Shirt Size

Abstract

Describe your talk in 500 words or less

☐ I'm actually available the date of the talk

SUBMIT BUTTONS

Finally, let's finish off our form with a submit button. The `<button>` element represents a button that will submit its containing `<form>`:

```
<div class='form-row'>  
  <button>Submit</button>  
</div>
```

Clicking the button tells the browser to validate all of the `<input/>` elements in the form and submit it to the `action` URL if there aren't any validation problems. So, you should now be able to type in something that's not an email address into our email field, click the `<button>`, and see an error message.



Name

Email

Type of Talk ☐ Main Stage

T-Shirt Size

! Please include an '@' in the email address. 'notanemail' is missing an '@'.

This also gives us a chance to see how the user's input gets sent to the server. First, enter some values into all the `<input/>` fields, making sure the email address validates correctly. Then, click the button and inspect the resulting URL in your browser. You should see something like this:

```
speaker-submission.html?full-name=Rick&email=rick%40internetingishard.com&talk-
```

Everything after the `?` represents the variables in our form. Each `<input/>`'s `name` attribute is followed by an equal sign, then its value, and each variable is separated by an `&` character. If we had a backend server, it'd be pretty easy for it to pull out all this information, query a database (or whatever), and let us know whether the form submission was successful or not.



we had some experience styling buttons in the [pseudo-classes](#) section of the [CSS Selectors](#) chapter. Back then, we were applying these styles to an `<a>` element, but we can use the same techniques on a `<button>`.

☐ I'm actually available the date of the talk

Submit

Clean up that ugly default `<button>` styling by adding the following to our stylesheet:

```
.form-row button {  
  font-size: 16px;  
  font-weight: bold;  
  
  color: #FFFFFF;  
  background-color: #5995DA;  
  
  border: none;  
  border-radius: 3px;  
  
  padding: 10px 40px;  
  cursor: pointer;  
}  
  
.form-row button:hover {
```



```
.form-row button:active {  
  background-color: #407FC7;  
}
```

As with our checkbox, we need to take that 120px label column into account, so include one more rule inside our media query:

```
@media only screen and (min-width: 700px) {  
  /* ... */  
  .form-row button {  
    margin-left: 120px;  
  }  
}
```

SUMMARY

In this chapter, we introduced the most common HTML form elements. We now have all these tools for collecting input from our website visitors:

- `<input type='text'/>`
- `<input type='email'/>`
- `<input type='radio'/>`
- `<select>` and `<option>`
- `<textarea>`



- `<button>`

You should be pretty comfortable with the HTML and CSS required to build beautiful forms, but actually making these forms functional requires some skills we don't have yet. That stuff is out of scope for this tutorial, but it might help to have some context. Generally speaking, there are two ways to process forms:

- Use the `action` attribute to send the form data to a backend URL, which then redirects to a success or error page. We got a little glimpse of this in the previous section, and it doesn't require any JavaScript.
- Use AJAX queries to submit the form without leaving the page. Success or error messages are displayed on the same page by manipulating the HTML with JavaScript.

Depending on how your organization is structured, form processing may not be part of your job role as a frontend web developer. If that's the case, you'll need to coordinate closely with a backend developer on your team to make sure the `<form>` submits the correct name-value pairs. Otherwise, it'll be up to you to make sure the frontend and backend of your forms fit neatly together.

Next, we have our final chapter in *HTML & CSS Is Hard*. We'll round out our frontend skills with a thorough discussion of web fonts and practical typographic principles that every web developer should know about.

NEXT CHAPTER >



InternetingIsHard.com is an independent publisher of premium web development tutorials. All content is authored and maintained by Oliver James. He loves hearing from readers, so [come say hello!](#)



More tutorials are coming. (Lots more.)

Enter your email , and we'll let you know when they get here.

© 2017 INTERNETINGISHARD.COM

[CONTACT](#)[LICENSING](#)[PRIVACY](#)[TERMS](#)