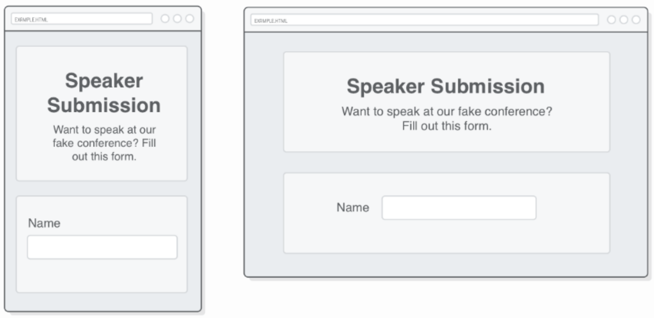
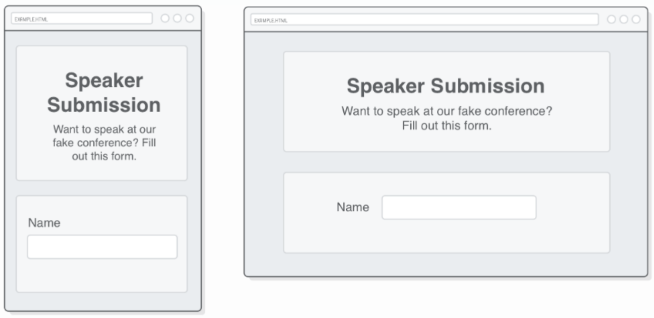
Check out that awesome use of the `flex-direction` property to make the `<label>` appear on top of its `<input/>` element in the mobile layout, but to the left of it in the desktop layout.



Speaker Submission

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



Speaker Submission

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

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'
         placeholder='joe@example.com'/>
</div>
```

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

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'
```

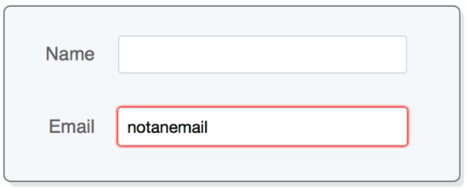
```
        type='email'
        placeholder='joe@example.com'/>
</div>
```

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

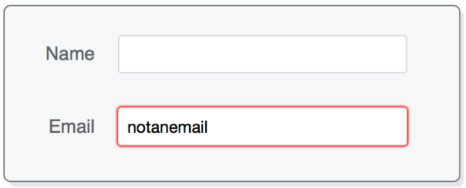
The `<input/>` element's `type` attribute also lets you do basic input validation. For example, let's try adding another input element that *only* accepts email addresses instead of arbitrary text values:

```
<div class='form-row'>
  <label for='email'>Email</label>
  <input id='email'
        name='email'
        type='email'
        placeholder='joe@example.com'/>
</div>
```

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.



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
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.



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.

There's a bunch of other built-in validation options besides email addresses, 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, 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.

There's a bunch of other built-in validation options besides email addresses, 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;
  /* ... */
}
```

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.

```
/* 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.

Let's not forget about our desktop styles. Update the corresponding `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;
  }
  /* ... */
}
```

Let's not forget about our desktop styles. Update the corresponding `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;
  box-shadow: none; /* Remove default red glow in Firefox */
}
```

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.

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;
  box-shadow: none; /* Remove default red glow in Firefox */
}
```

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.

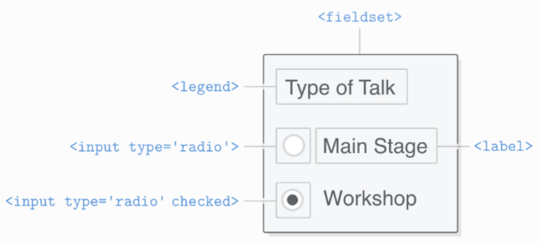
both the border and the text with a custom shade of red when the user entered an unacceptable value. Add the following rule to our stylesheet, outside of the media query:

```
.form-row input[type='text']:invalid,
.form-row input[type='email']:invalid {
  border: 1px solid #D55C5F;
  color: #D55C5F;
  box-shadow: none; /* Remove default red glow in Firefox */
}
```

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 `<fieldset>` and `<legend>` elements are for. Every radio button group you 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'
```

which is why each one of them needs an explicit `value` attribute. This is the value that will get sent to the server when the user submits the form. It's also very 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-`