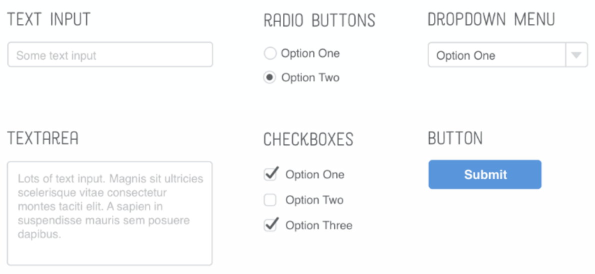
HTML FORMS

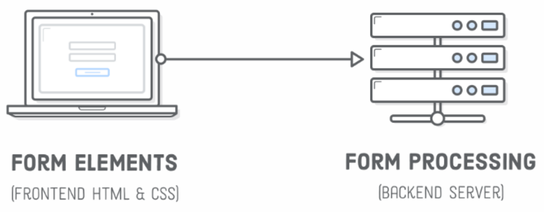
Nº 13. OF HTML & CSS IS HARD

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.



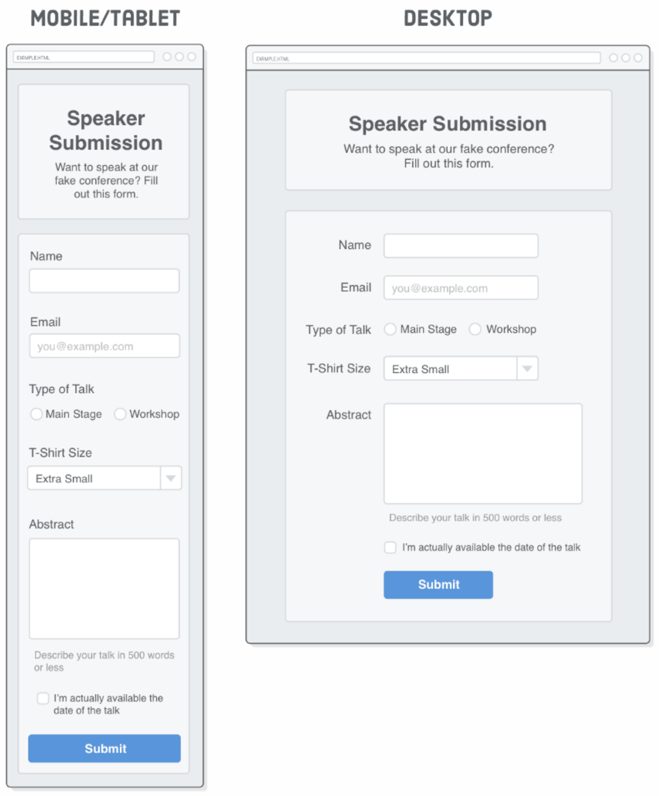
Forms are the "money pages." They're how e-commerce sites sell their products, how SaaS companies collect payment for their service, and how non-profit groups raise money online. Many companies measure the success of their website by the effectiveness of its forms because they answer questions like "how 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.



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
DESKTOP
Speaker Submission
Want to speak at our fake conference? Fill out this form.
Name
Email
you@example.com
Type of Talk
Main Stage
Workshop
T-Shirt Size
Extra Small
Abstract
Describe your talk in 500 words or less
I'm actually available the date of the talk
Submit

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.

Create a new Atom project called `forms` and stick a new HTML file in it called `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>
```

called `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 called `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;
}
    <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;
}

body {
  color: #5D6063;
  background-color: #EAEDF0;
    <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;
}

body {
  color: #5D6063;
```

```css
  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;
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  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;

}

body {
  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;
}

body {
  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;
  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;
}
  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;
}
```

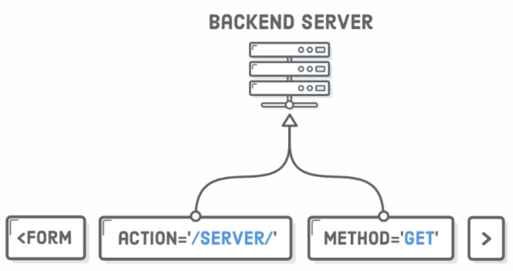
Notice that we're adhering to the mobile-first development approach that we discussed in the *Responsive Design* chapter. These base CSS rules give us our mobile layout and provide a foundation for the desktop layout, too. We'll create the media query for a fixed-width desktop layout later in the chapter.

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>`:

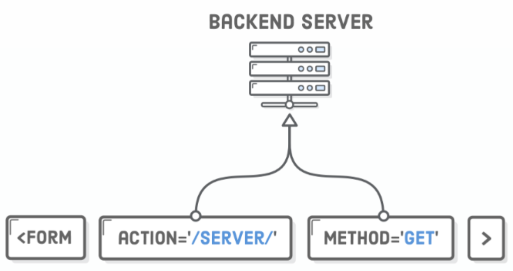
```
<form action='' method='get' class='speaker-form'>
</form>
```

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.



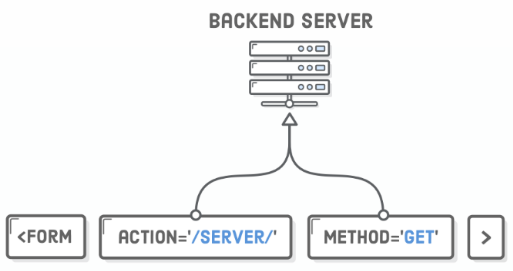
The `method` attribute can be either `post` or `get`, both of which define how the form is submitted to the backend server. This is largely dependent on how your web server wants to handle the form, but the general rule of thumb is to use `post` when you're *changing* data on the server, reserving `get`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

be focusing on the frontend in this chapter.



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
be focusing on the frontend in this chapter.



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;
  margin: 0 0 40px 0;
}
```

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;
  margin: 0 0 40px 0;
}
```

```
.speaker-form {
  background-color: #F6F7F8;
  border: 1px solid #D6D9DC;
  border-radius: 3px;

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

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

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>
```

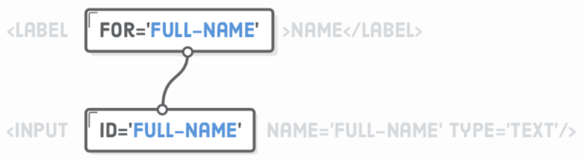
First, we have a container `<div>` to help with styling. This is pretty common for separating input elements. Second, we have a `<label>`, which you can think of as another semantic HTML element, like `<article>` or `<figcaption>`, but for form labels. A label's `for` attribute must match the

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:

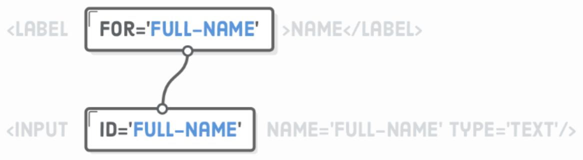
```
<div class='form-row'>
  <label for='full-name'>Name</label>
  <input id='full-name' name='full-name' type='text'/>
</div>
```

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.



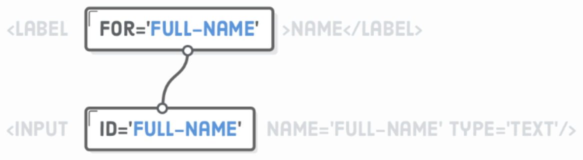
```
<div class='form-row'>
  <label for='full-name'>Name</label>
  <input id='full-name' name='full-name' type='text'/>
</div>
```

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.



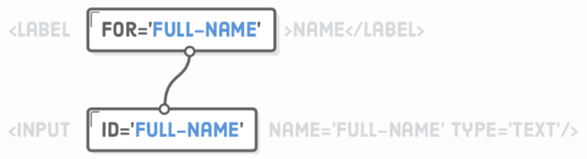
Third, the `<input/>` element creates a text field. It's a little different from other elements we've encountered because it can dramatically change appearance depending on its `type` attribute, but it always creates some kind of interactive user input. We'll see other values besides `text` throughout the

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.

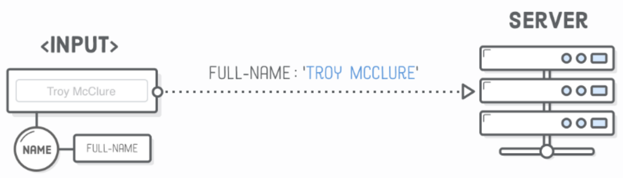


Third, the `<input/>` element creates a text field. It's a little different from other elements we've encountered because it can dramatically change appearance depending on its `type` attribute, but it always creates some kind of interactive user input. We'll see other values besides `text` throughout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

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.



Third, the `<input/>` element creates a text field. It's a little different from other 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
Third, the `<input/>` element creates a text field. It's a little different from other 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.

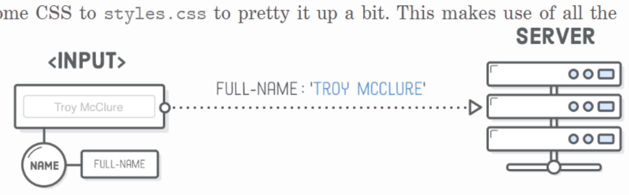
Conceptually, an `<input/>` element represents a "variable" that gets sent to the backend server. The `name` attribute defines the name of this variable, and the value is whatever the user entered into the text field. Note that you can pre-populate this value by adding a `value` attribute to an `<input/>`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

Third, the `<input/>` element creates a text field. It's a little different from other 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



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:

```css
.form-row {
  margin-bottom: 40px;
  display: flex;
  justify-content: flex-start;
  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;
```

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.

All of our styles are "namespaced" in a `.form-row` descendant selector. Isolating `<input/>` and `<label>` styles like this makes it easier to create different kinds of forms. We'll see why it's convenient to avoid global `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.

```css
@media only screen and (min-width: 700px) {
  .speaker-form-header,
  .speaker-form {
    width: 600px;
  }
  .form-row {
```