

HTML. Unit 9. Forms.

[1. Introduction](#)

[2. What are web forms?](#)

[3. Basic native form controls](#)

[3.1. The <label> element](#)

[3.1.1. Labels are clickable, too!](#)

[3.2. Text input fields](#)

[3.2.1. Single line text fields](#)

[3.2.1.1. Password field](#)

[3.2.2. Multiple line text fields](#)

[3.2.3. Default values in text fields](#)

[3.3. Checkable items: checkboxes and radio buttons](#)

[3.3.1. Checkbox](#)

[3.3.2. Radio button](#)

[3.4. The <select> element](#)

[3.5. File picker](#)

[3.6. Buttons](#)

[3.7. Proposed exercise: Native controls](#)

[4. HTML5 input types](#)

[4.1. E-mail address field](#)

[4.2. URL field](#)

[4.3. Phone number field](#)

[4.4. Numeric field](#)

[4.5. Slider controls](#)

[4.6. Date and time pickers](#)

[4.7. Color picker control](#)

[4.8. Search field](#)

[4.9. Proposed exercise: HTML5 input types](#)

[5. Your first “real” form](#)

[5.1. Designing your form](#)

[5.2. Implementing your form](#)

[5.3. The <form> element](#)

[5.4. The <label>, <input> and <textarea> elements](#)

[5.5. The <button> element](#)

[5.6. Sending form data to your web server](#)

[5.6.1. Client-side form validation](#)

[5.6.1.1. Further reading](#)

[5.6.2. On the server side: retrieving the data](#)[5.6.3. The GET method](#)[5.6.3.1. Proposed exercise: Contact form](#)[5.6.3.2. Proposed exercise: Full contact form](#)[5.6.3.3. Proposed exercise: Greetings form](#)[5.6.3.4. Proposed exercise: Full greetings form](#)[5.6.4. The POST method](#)[5.6.4.1. Proposed exercise: Login form](#)[5.6.4.2. Proposed exercise: Full login form](#)

1. Introduction

This unit provides some instructions and examples that will help you to learn the essentials of web forms. Web forms are a very powerful tool for interacting with users — most commonly they are used for collecting data from users, or allowing them to control a user interface. However, for historical and technical reasons it's not always obvious how to use them to their full potential. In the sections listed below, we'll cover all the essential aspects of Web forms including marking up their HTML structure, validating form data, and submitting data to the server.

2. What are web forms?

Web forms are one of the main points of interaction between a user and a web site or application. Forms allow users to enter data, which is generally sent to a web server for processing and storage, or used on the client side to immediately update the interface in some way (for example, add another item to a list, or show or hide a UI feature).

A web form's HTML is made up of one or more **form controls** (sometimes called **widgets**), plus some additional elements to help structure the overall form — they are often referred to as **HTML forms**. The controls can be single or multi-line text fields, dropdown boxes, buttons, checkboxes, or radio buttons, and are mostly created using the `<input>` element, although there are some other elements to learn about too.

Form controls can also be programmed to enforce specific formats or values to be entered (**form validation**), and paired with text labels that describe their purpose to both sighted and blind users.

3. Basic native form controls

In the next sections we will mark up several functional web form examples, using some form controls and common structural elements, and focusing on accessibility best practices. Now we will look at the functionality of the different form controls, or widgets, in detail — studying all the different options available to collect different types of data. In this particular section we will look at the original set of form controls, available in all browsers since the early days of the web.

3.1. The `<label>` element

The `<label>` element is the formal way to define a label for an HTML form widget. This is the most important element if you want to build accessible forms. When implemented properly, screen readers will speak a form element's label along with any related instructions, as well as being useful for sighted users. Take this example, where we nest the form control within the `<label>`, implicitly associating it:

```
1. <label>
2.   Name: <input type="text" name="name" />
3. </label>
```

With the `<label>` associated correctly with the `<input>` a screen reader will read out something like “Name, edit text”. If there is no label, or if the form control is neither implicitly or explicitly associated with a label, a screen reader will read out something like “Edit text blank”, which isn't very helpful at all.

3.1.1. Labels are clickable, too!

Another advantage of properly setting up labels is that you can click or tap the label to activate the corresponding widget. This is useful for controls like text inputs, where you can click the label as well as the input to focus it, but it is especially useful for radio buttons and checkboxes. The hit area of such a control can be very small, so it is useful to make it as easy to activate as possible.

For example, clicking on the labels “I like cherry” or “I like banana” in the example below will toggle the selected state of the *cherry* or *banana* checkboxes respectively:

```
1. <label>
2.   <input type="checkbox" name="cherry" value="cherry" />
3.   I like cherry
4. </label><br />
5. <label>
6.   <input type="checkbox" name="banana" value="banana" />
7.   I like banana
8. </label><br />
```

3.2. Text input fields

Text `<input>` fields are the most basic form widgets. They are a very convenient way to let the user enter any kind of data because they can take many different forms depending on its `type` attribute value. It is used for creating most types of form widgets including single line text fields, time and date controls, controls without text input like checkboxes, radio buttons, and color pickers, and buttons too.

All basic text controls share some common behaviors:

- They can be marked as `required`, to specify that a form field needs to be filled in before the form can be submitted.
- They can be marked as `readonly` (the user cannot modify the input value but it is still sent with the rest of the form data) or `disabled` (the input value can't be modified and is never sent with the rest of the form data).
- They can have a `placeholder`. This is text that appears inside the text input box that should be used to briefly describe the purpose of the box.
- They can be constrained in `size` (the physical size of the box) and `minlength` and `maxlength` (the minimum and maximum number of characters that can be entered into the box).
- They can benefit from spell checking (using the `spellcheck` attribute), if the browser supports it.

Keep in mind that HTML form text fields are simple plain text input controls. This means that you cannot use them to perform [rich editing](#) (bold, italic, etc.). All rich text editors you'll encounter are custom widgets created with HTML, CSS, and JavaScript.

3.2.1. Single line text fields

A single line text field is created using an `<input>` element whose `type` attribute value is set to `text`, or by omitting the `type` attribute altogether (`text` is the default value). The value `text` for this attribute is also the fallback value if the value you specify for the `type` attribute is unknown by the browser (for example if you specify `type="color"` and the browser doesn't support native color pickers).

Let's see this example using a couple of single line text fields:

```
1. <label>
2.   Name (5 to 10 characters):
3.   <input type="text" name="name" required
4.       minlength="5" maxlength="10" size="15"
5.       placeholder="e.g. Fernando">
6. </label><br />
7. <label>
8.   Comment:
9.   <input type="text" name="comment" required
10.      placeholder="e.g. I like this example">
11. </label><br />
```

HTML5 enhanced the basic original single line text field by adding special values for the `type` attribute that enforce specific validation constraints and other features, for example specific to entering emails, URLs or numbers. We'll cover those in a section below (The HTML5 input types).

3.2.1.1. Password field

One of the original input types was the `password` text field type:

```
1. <label>
2.   Password: <input type="password" name="password">
3. </label>
```

The `password` value doesn't add any special constraints to the entered text, but it does obscure the value entered into the field (e.g. with dots or asterisks) so it can't be easily read by others.

Keep in mind this is just a user interface feature; unless you submit your form securely, it will be sent in plain text, which is bad for security — a malicious party could intercept your data and steal passwords, credit card details, or whatever else you've submitted. The best way to protect users from this is to host any pages involving forms over a secure connection (i.e. at an `https://...` address), so the data is encrypted before it is sent.

Browsers recognize the security implications of sending form data over an insecure connection, and have warnings to deter users from using insecure forms. For more information on what Firefox implements, see [Insecure passwords](#).

3.2.2. Multiple line text fields

The HTML `<textarea>` element represents a multi-line plain-text editing control, useful when you want to allow users to enter a sizeable amount of free-form text, for example a comment on a review or feedback form:

```
1. <label>Tell us your story:
2.   <textarea name="story" rows="5">
3.     It was a dark and stormy night...
4.   </textarea>
5. </label>
```

You can use the `rows` and `cols` attributes to specify an exact size for the `<textarea>` to take. Setting these sometimes is a good idea for consistency, as browser defaults can differ. We also are using a default content (entered between the opening and closing tags), since `<textarea>` does not support the `value` attribute.

The `<textarea>` element also accepts several attributes common to form `<input>` element, such as `autocomplete`, `autofocus`, `disabled`, `placeholder`, `readonly`, and `required`.

3.2.3. Default values in text fields

Notice that the `<input>` tag is an empty element, meaning that it doesn't need a closing tag. The `<textarea>` element however must be closed with the proper ending tag. This has an impact on a

specific feature of forms: the way you define the default value. To define the default value of an `<input>` element you have to use the `value` attribute like this:

```
1. <input type="text" value="by default this element is filled with this text">
```

On the other hand, if you want to define a default value for a `<textarea>`, you put it between the opening and closing tags of the `<textarea>` element, like this:

```
1. <textarea>
2.   by default this element is filled with this text
3. </textarea>
```

3.3. Checkable items: checkboxes and radio buttons

Checkable items are controls whose state can be changed by clicking on them or their associated labels. There are two kinds of checkable item: the check box and the radio button. Both use the `checked` attribute to indicate whether the widget is checked by default or not.

It's worth noting that these widgets do not behave exactly like other form widgets. For most form widgets, once the form is submitted all widgets that have a `name` attribute are sent, even if no value has been filled out. In the case of checkable items, their values are sent only if they are checked. If they are not checked, nothing is sent, not even their name. If they are checked but have no value, the name is sent with a value of *on*.

3.3.1. Checkbox

A check box is created using the `<input>` element with a `type` attribute set to the value [checkbox](#). Elements of type `checkbox` are rendered by default as boxes that are checked (ticked) when activated, like you might see in an official government paper form. The exact appearance depends upon the operating system configuration under which the browser is running. Generally this is a square but it may have rounded corners. A checkbox allows you to select single values for submission in a form (or not).

Let's see and try a very simple example:

```
1. <label>
2.   <input type="checkbox" name="carrots" value="carrots" checked />
3.   Do you like carrots?
4. </label>
```

Including the `checked` attribute makes the checkbox checked automatically when the page loads. Clicking the checkbox or its associated label toggles the checkbox on and off.

Due to the on-off nature of checkboxes, the checkbox is considered a toggle button. Many developers and designers are expanding the default checkbox styling to create buttons that look like toggle switches.

3.3.2. Radio button

A radio button is created using the `<input>` element with its `type` attribute set to the value `radio`. Elements of type `radio` are generally used in **radio groups** (collections of radio buttons describing a set of related options). Only one radio button in a given group can be selected at the same time. Radio buttons are typically rendered as small circles, which are filled or highlighted when selected.

Let's see a simple example containing several radio buttons and how a browser may render it:

```
1. What is your favorite meal?<br />
2. <label>
3.   <input type="radio" name="meal" value="soup" checked />Soup
4. </label><br />
5. <label>
6.   <input type="radio" name="meal" value="curry" />Curry
7. </label><br />
8. <label>
9.   <input type="radio" name="meal" value="pizza" />Pizza
10. </label><br />
```

As seen above, several radio buttons can be tied together. If they share the same value for their `name` attribute, they will be considered to be in the same group of buttons. Only one button in a given group may be checked at a time; this means that when one of them is checked all the others automatically get unchecked. When the form is sent, only the value of the checked radio button is sent. If none of them are checked, the whole pool of radio buttons is considered to be in an unknown state and no

value is sent with the form. Once one of the radio buttons in a same-named group of buttons is checked, it is not possible for the user to uncheck all of the buttons without resetting the form.

3.4. The `<select>` element

The HTML `<select>` element represents a control that provides a menu of options. For example:

```
1.  <label>Choose the pet you most like:
2.    <select name="pets" id="pet-select">
3.      <option value="">--Please choose an option--</option>
4.      <option value="dog">Dog</option>
5.      <option value="cat">Cat</option>
6.      <option value="hamster">Hamster</option>
7.      <option value="parrot">Parrot</option>
8.      <option value="spider">Spider</option>
9.      <option value="goldfish">Goldfish</option>
10.    </select>
11.  </label>
```

The above example shows typical `<select>` usage. It is associated with a `<label>` for accessibility purposes, as well as a `name` attribute to represent the name of the associated data submitted to the server. Each menu option is defined by an `<option>` element nested inside the `<select>`.

Each `<option>` element should have a `value` attribute containing the data value to submit to the server when that option is selected. If no value attribute is included, the value defaults to the text contained inside the element. You can include a `selected` attribute on an element to make it selected by default when the page first loads.

The `<select>` element has some unique attributes you can use to control it, such as `multiple` to specify whether multiple options can be selected, and `size` to specify how many options should be shown at once. It also accepts most of the general form input attributes such as `required`, `disabled`, `autofocus`, etc.

3.5. File picker

There is one last `<input>` type that came to us in early HTML: the file input type. Forms are able to send files to a server (this specific action is also detailed in the [Sending form data](#) article). The file picker widget can be used to choose one or more files to send.

To create a [file picker widget](#), you can use the `<input>` element with its `type` attribute set to `file`. The types of files that are accepted can be constrained using the `accept` attribute. In addition, if you want to let the user pick more than one file, you can do so by adding the `multiple` attribute.

In the following example, a file picker is created to request graphic image files. The user is allowed to select multiple files in this case:

```
1. <input type="file" name="file" accept="image/*" multiple />
```

On some mobile devices, the file picker can access photos, videos, and audio captured directly by the device's camera and microphone by adding capture information to the `accept` attribute like so:

```
1. <input type="file" accept="image/*;capture=camera">
2. <input type="file" accept="video/*;capture=camcorder">
3. <input type="file" accept="audio/*;capture=microphone">
```

3.6. Buttons

The HTML `<button>` element represents a clickable button, used to submit forms or anywhere in a document for accessible, standard button functionality. By default, HTML buttons are presented in a style resembling the platform the browser runs on, but you can change buttons' appearance with CSS.

The default behavior of the button can be changed with the `type` attribute. Possible values are:

- `submit` : The button submits the form data to the server. This is the default if the attribute is not specified for buttons associated with the form or if the attribute contains an empty or an invalid value.

- `reset` : The button resets all the controls to their initial values. You should use it only when necessary, since this behavior tends to annoy users.
- `button` : The button has no default behavior, and does nothing when pressed by default. It can have client side scripts listen to the element's events, which are triggered when the events occur.

Let's see all types of buttons with a simple example:

```
1.  <p>
2.    <label>Enter your comment: <input type="text" name="comment" required />
    </label>
3.  </p>
4.  <p>
5.    <button type="submit">This is a submit button</button>
6.  </p>
7.  <p>
8.    <button type="reset">This is a reset button</button>
9.  </p>
10. <p>
11.   <button type="button">This is a simple button</button>
12. </p>
```

As you can see from the examples, `<button>` elements let you use HTML in their content, which is inserted between the opening and closing `<button>` tags. `<input>` elements on the other hand are empty elements; their displayed content is inserted inside the `value` attribute, and therefore only accepts plain text as content.

3.7. Proposed exercise: Native controls

Create a web page to show samples of all the input elements in this section: single line and multiple line text, password, checkboxes and radio buttons, select and file picker. You must include at least two examples of each of them. You have to use paragraphs and labels, and also the “required” attribute and all necessary field constraints have to be set for all of them. Check the result in your browser, and do not forget to include all basic HTML tags and validate your code. Finally, upload the code to your domain and check the result in your mobile phone.

— Put all the tags inside a `<form>` container and use a submit button so that you can check that the fields are properly validated:

```
1. <form>
2.   <p><label>
3.     Name: <input type="text" name="name" required />
4.   </label></p>
5.   <p><label>
6.     Surname: <input type="text" name="surname" required />
7.   </label></p>
8.   <p><label>
9.     Password: <input type="password" name="password1" required />
10.  </label></p>
11.  <p><label>
12.    Repeat your password: <input type="password" name="password2" required />
13.  </label></p>
14.    ...
15.  <p><button>Submit</button></p>
16. </form>
```

4. HTML5 input types

In the previous section we looked at the `<input>` element, covering the original values of the `type` attribute available since the early days of HTML. Now we'll look at the functionality of newer form controls in detail, including some new input types, which were added in HTML5 to allow collection of specific types of data.

4.1. E-mail address field

This type of field is set using the value `email` for the `type` attribute:

```
1. <label>
2.   Enter a valid email:
3.   <input type="email" name="email" placeholder="e.g. test@test.com" required />
4. </label>
```

When this `type` is used, the user is required to type a valid email address into the field. Any other content causes the browser to display an error when the form is submitted. You can see this in action [here](#):

You can also use the `multiple` attribute in combination with the `email` input type to allow several email addresses to be entered in the same input (separated by commas):

```
1. <label>
2.   Multiple emails: <input type="email" name="emails" multiple />
3. </label>
```

On some devices (notably touch devices with dynamic keyboards like smart phones) a different virtual keypad might be presented that is more suitable for entering email addresses, including the `@` key. This is another good reason for using these newer input types, improving the user experience for users of these devices.

4.2. URL field

A special type of field for entering URLs can be created using the value `url` for the `type` attribute:

```
1. <label>
2.   Enter URL:
3.   <input type="url" name="url" placeholder="e.g. https://..." required />
4. </label>
```

It adds special validation constraints to the field. The browser will report an error if no protocol (such as `http:` is entered, or if the URL is otherwise malformed. You can see this in action [here](#):

On devices with dynamic keyboards, the default keyboard will often display some or all of the colon, period, and forward slash as default keys.

4.3. Phone number field

A special field for filling in phone numbers can be created using `tel` as the value of the `type` attribute:

```
1. <label>
2.     Enter phone number:
3.     <input type="tel" name="tel" placeholder="e.g. 123 456 789" />
4. </label>
```

When accessed via a touch device with a dynamic keyboard, most devices will display a numeric keypad when `type="tel"` is encountered, meaning this type is useful whenever a numeric keypad is useful, and doesn't just have to be used for telephone numbers.

Due to the wide variety of phone number formats around the world, this type of field does not enforce any constraints on the value entered by a user (this means it may include letters, etc.).

4.4. Numeric field

Controls for entering numbers can be created with an `<input type="number">`. This control looks like a text field but allows only floating-point numbers, and usually provides buttons in the form of a spinner to increase and decrease the value of the control. On devices with dynamic keyboards, the numeric keyboard is generally displayed.

With the `number` input type, you can constrain the minimum and maximum values allowed by setting the `min` and `max` attributes. You can also use the `step` attribute to set the increment increase and decrease caused by pressing the spinner buttons. By default, the number input type only validates if the number is an integer. To allow float numbers, specify `step="any"`. If omitted, the `step` value defaults to `1`, meaning only whole numbers are valid.

Let's look at some examples. The first one below creates a number control whose value is restricted to any value between `1` and `10`, and whose increase and decrease buttons change its value by `2`:

```
1. <input type="number" name="age" min="1" max="10" step="2" value="1" />
```

The second one creates a number control whose value is restricted to any value between `0` and `1` inclusive, and whose increase and decrease buttons change its value by `0.01`:

```
1. <input type="number" name="change" min="0" max="1" step="0.01" value="0" />
```

The `<input type="number">` makes sense when the range of valid values is limited, for example a person's age or height. If the range is too large for incremental increases to make sense (such as USA ZIP codes, which range from 00001 to 99999), the `<input type="tel">` might be a better option; it provides the numeric keypad while forgoing the number's spinner UI feature.

4.5. Slider controls

Another way to pick a number is to use a **slider**. You see these quite often on sites like house buying sites where you want to set a maximum property price to filter by. Let's look at a live example to illustrate this:

Usage-wise, sliders are less accurate than text fields. Therefore, they are used to pick a number whose *precise* value is not necessarily important.

A slider is created using the `<input>` with its `type` attribute set to the value `range` (`<input type="range">`). The slider-thumb can be moved via mouse or touch, or with the arrows of the keypad. It's important to properly configure your slider. To that end, it's highly recommended that you set the `min`, `max`, and `step` attributes which set the minimum, maximum and increment values, respectively.

Let's look at the code behind the above example, so you can see how its done. First of all, the basic HTML:

```
1. <form>
2.   <p>Choose a maximum house price:</p>
3.   <input type="range" name="range"
4.       min="50000" max="500000" step="100" value="250000"
5.       oninput="number.value = this.value" />
6.   <input type="number" name="number"
7.       min="50000" max="500000" step="100" value="250000"
8.       oninput="range.value = this.value" />
9. </form>
```

This example creates a slider whose value may range between 50000 and 500000, which increments/decrements by 100 at a time. We've given it default value of 250000, using the `value` attribute.

One problem with sliders is that they don't offer any kind of visual feedback as to what the current value is. This is why we've included an `<input type="number">` element to contain the current value using some JavaScript code (we will go into this in a future unit).

4.6. Date and time pickers

Gathering date and time values has traditionally been a nightmare for web developers. For a good user experience, it is important to provide a calendar selection UI, enabling users to select dates without necessitating context switching to a native calendar application or potentially entering them in differing formats that are hard to parse. For example, the last minute of the previous millennium can be expressed in many different ways: 1999/12/31 , 23:59 , 12/31/99T11:59PM , etc.

HTML date controls are available to handle this specific kind of data, providing calendar widgets and making the data uniform.

A date and time control is created using the `<input>` element and an appropriate value for the `type` attribute, depending on whether you wish to collect dates, times, or both. Let's look at the different available types in brief:

```
1. <p><label>
2.   Local date time: <input type="datetime-local" name="datetime" />
3. </label></p>
4. <p><label>
5.   Month: <input type="month" name="month">
6. </label></p>
7. <p><label>
8.   Time: <input type="time" name="time">
9. </label></p>
10. <p><label>
11.   Week: <input type="week" name="week">
12. </label></p>
```

All date and time controls can be constrained using the `min` and `max` attributes, with further constraining possible via the `step` attribute (whose value varies according to input type):

```
1. <label>
2.   When is your birthday?
```



```
3. <input type="date" name="date" min="1975-01-01" max="2025-12-31" step="1" />
4. </label>
```

4.7. Color picker control

Colors are always a bit difficult to handle. There are many ways to express them: RGB values (decimal or hexadecimal), HSL values, keywords, and so on.

A color control can be easily created using the `<input>` element with its `type` attribute set to the value `color`. For example:

```
1. <label>
2.     Select color: <input type="color" name="color" />
3. </label>
```

When supported, clicking a color control will tend to display the operating system's default color picking functionality for you to actually make your choice with. Here is a live example for you to try out:

4.8. Search field

Search fields are intended to be used to create search boxes on pages and apps. This type of field is set by using the value `search` for the `type` attribute:

```
1. <input type="search" name="search" placeholder="Search" required />
```

The main difference between a `text` field and a `search` field is how the browser styles its appearance. Often, `search` fields are rendered with rounded corners; they also sometimes display an “ⓧ”, which clears the field of any value when clicked. Additionally, on devices with dynamic keyboards, the keyboard's enter key may read “search”, or display a magnifying glass icon.

Another worth-noting feature is that the values of a `search` field can be automatically saved and re-used to offer auto-completion across multiple pages of the same website; this tends to happen automatically in most modern browsers.

4.9. Proposed exercise: HTML5 input types

Create a web page to show samples of all the input elements in this section: email, url, phone number, numeric field, slide control, date and time, color picker and search field. You must include at least two examples for each of them. You have to use paragraphs and labels, and also the “required” attribute and all necessary field constraints have to be set for all of them. Check the result in your browser, and do not forget to include all basic HTML tags and validate your code. Finally, upload the code to your domain and check the result in your mobile phone.

— Put all the tags inside a `<form>` container and use a submit button so that you can check that the fields are properly validated:

```
1.  <form>
2.    <p><label>
3.      Primary email: <input type="email" name="email1" required />
4.    </label></p>
5.    <p><label>
6.      Secondary email: <input type="email" name="email2" required />
7.    </label></p>
8.    <p><label>
9.      Your own website: <input type="url" name="website1" required />
10.   </label></p>
11.   <p><label>
12.     Your school's website: <input type="url" name="website2" required />
13.   </label></p>
14.   ...
15.   <p><button>Submit</button></p>
16. </form>
```

5. Your first “real” form

The section provides you with your very first experience of creating a real web form, including designing a simple form, implementing it using the right HTML form controls and other HTML elements, and describing how data is sent to a server. We’ll expand on each of these subtopics in more detail later.

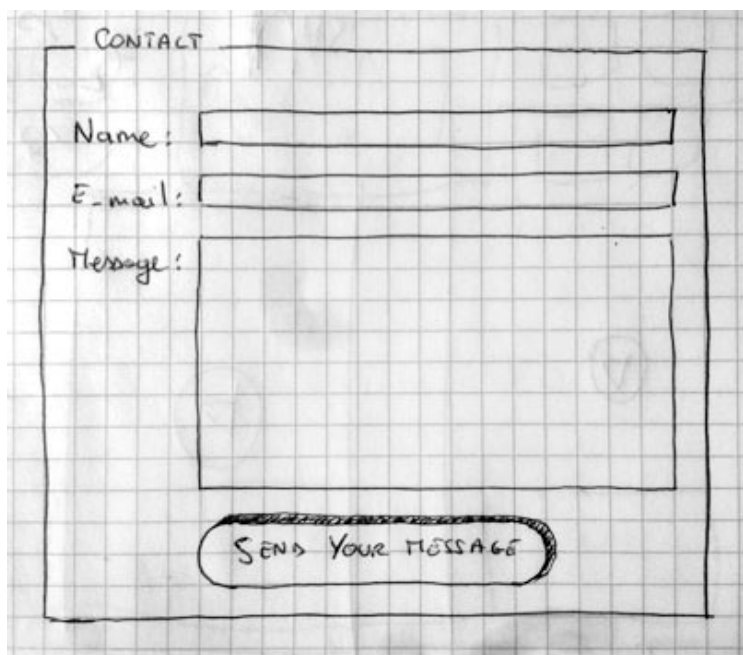
5.1. Designing your form

Before starting to code, it's always better to step back and take the time to think about your form. Designing a quick mockup will help you to define the right set of data you want to ask your user to enter. From a user experience (UX) point of view, it's important to remember that the bigger your form, the more you risk frustrating people and losing users. Keep it simple and stay focused: ask only for the data you absolutely need.

Designing forms is an important step when you are building a site or application. It's beyond the scope of this guide to cover the user experience of forms, but if you want to dig into that topic you should read the following articles:

- Smashing Magazine has some [good articles about forms UX](#), including an older but still relevant [Extensive Guide To Web Form Usability](#) article.
- UXMatters is also a very thoughtful resource with good advice from [basic best practices](#) to complex concerns such as [multi-page forms](#).

In this section, we'll build a simple contact form. Let's make a rough sketch:



A hand-drawn sketch of a contact form on graph paper. The form is titled "CONTACT" at the top left. It contains three input fields: "Name:" followed by a rectangular box, "E-mail:" followed by a rectangular box, and "Message:" followed by a larger rectangular box. At the bottom of the form is a rounded rectangular button labeled "SEND YOUR MESSAGE".

Our form will contain three text fields and one button. We are asking the user for their name, their e-mail and the message they want to send. Hitting the button will send their data to a web server.

5.2. Implementing your form

Let's start creating the HTML for our form. We will use the following HTML elements: `<form>` , `<label>` , `<input>` , `<textarea>` , `<button>` .

5.3. The `<form>` element

All forms start with a `<form>` element, like this:

```
1. <form action="contact.php" method="get">
2.
3. </form>
```

This element formally defines a form. It's a container element and it also supports some specific attributes to configure the way the form behaves. All of its attributes are optional, but it's standard practice to always set at least the `action` and `method` attributes:

- The `action` attribute defines the location (URL) where the form's collected data should be sent when it is submitted.
- The `method` attribute defines which HTTP method to send the data with (usually `GET` or `POST`).

We'll look at how those attributes work in our sending form data section later on.

5.4. The `<label>`, `<input>` and `<textarea>` elements

Our contact form is not complex: the data entry portion contains three text fields, each with a corresponding `<label>` :

- The input field for the name is a [single-line text field](#).
- The input field for the e-mail is an [input of type email](#): a single-line text field that accepts only e-mail addresses.
- The input field for the message is a `<textarea>` : [a multiline text field](#).

In terms of HTML code we need something like the following to implement these form widgets:

```
1. <form action="contact.php" method="GET">
2.   <p>
3.     <label>Name: <input type="text" name="name" required /></label>
4.   </p>
5.   <p>
6.     <label>E-mail: <input type="email" name="email" required /></label>
7.   </p>
8.   <p>
9.     <label>Message: <textarea name="message" required></textarea></label>
10.  </p>
11. </form>
```

For usability and accessibility, we include an explicit label for each form control. There is great benefit to doing this — it associates the label with the form control, enabling mouse, trackpad, and touch device users to click on the label to activate the corresponding control, and it also provides an accessible name for screen readers to read out to their users.

On the `<input>` element, the most important attribute is the `type` attribute. This attribute is extremely important because it defines the way the `<input>` element appears and behaves:

- In our simple example, we use `<input type="text">` for the first input (the default value for this attribute). It represents a basic single-line text field that accepts any kind of text input.
- For the second input, we use `<input type="email">` which defines a single-line text field that only accepts a well-formed e-mail address. This turns a basic text field into a kind of “intelligent” field that will perform some validation checks on the data typed by the user. It also causes a more appropriate keyboard layout for entering email addresses (e.g. with an @ symbol by default) to appear on devices with dynamic keyboards, like smartphones.

Last but not least, note the syntax of `<input>` vs. `<textarea></textarea>`. This is one of the oddities of HTML. The `<input>` tag is an empty element, meaning that it doesn't need a closing tag. `<textarea>` is not an empty element, meaning it should be closed with the proper ending tag.

5.5. The <button> element

The markup for our form is almost complete; we just need to add a button to allow the user to send, or “submit”, their data once they have filled out the form. This is done by using the `<button>` element. We only need to add the following just above the closing `</form>` tag:

```
1. <button type="submit">Send your message</button>
```

As explained in a previous section, the `<button>` element also accepts a `type` attribute, with one of three values: `submit`, `reset`, or `button`:

- A click on a `submit` button (the default value) sends the form’s data to the web page defined by the `action` attribute of the `<form>` element.
- A click on a `reset` button resets all the form widgets to their default value immediately. From a UX point of view, this is considered bad practice, so you should avoid using this type of button unless you really have a good reason to include one.
- A click on a `button` button does... nothing! That sounds silly, but it’s amazingly useful for building custom buttons, since you can define their chosen functionality with JavaScript.

5.6. Sending form data to your web server

The last part, and perhaps the trickiest, is to handle form data on the server side. The `<form>` element defines where and how to send the data thanks to the `action` and `method` attributes.

We provide a `name` to each form control. The names are important on both the client and server side; they tell the browser which name to give each piece of data and, on the server side, they let the server handle each piece of data by name. The form data is sent to the server as name/value pairs.

To name the data in a form you need to use the `name` attribute on each form widget that will collect a specific piece of data. Let’s look at our contact form again:

```
1. <form action="contact.php" method="GET">
2.   <p>
3.     <label>Name: <input type="text" name="name" required /></label>
4.   </p>
5. </form>
```

```
6.      <label>E-mail: <input type="email" name="email" required /></label>
7.      </p>
8.      <p>
9.          <label>Message: <textarea name="message" required></textarea></label>
10.     </p>
11.     <p>
12.         <button type="submit">Send your message</button>
13.     </p>
14. </form>
```

In our example, the form will send 3 pieces of data named “name”, “email”, and “message”. That data will be sent to the URL “contact.php” using the HTTP GET method.

On the server side, the script at the URL “contact.php” will receive the data as a list of three key/value items contained in the HTTP request. The way this script will handle that data is up to you. Each server-side language (PHP, Python, Ruby, Java, C#, etc.) has its own mechanism of handling form data. It’s beyond the scope of this guide to go deeply into that subject for each language, but we will provide an example so that you can test your own forms using PHP.

5.6.1. Client-side form validation

Before submitting data to the server, it is important to ensure all required form controls are filled out, in the correct format. This is called **client-side form validation**, and helps ensure data submitted matches the requirements set for the various form controls.

Client-side validation is an initial check and an important feature of good user experience; by catching invalid data on the client-side, the user can fix it straight away. If it gets to the server and is then rejected, a noticeable delay is caused by a round trip to the server and then back to the client-side to tell the user to fix their data.

If you go to any popular site with a registration form, you will notice that they provide feedback when you don’t enter your data in the format they are expecting. You’ll get messages such as:

- “This field is required” (You can’t leave this field blank).
- “Please enter your phone number in the format xxxxxxxxxx” (A specific data format is required for it to be considered valid).

- “Please enter a valid email address” (the data you entered is not in the right format).
- “Your password needs to be between 8 and 30 characters long and contain one uppercase letter, one symbol, and a number” (A very specific data format is required for your data).

This is called **form validation**. When you enter data, the browser and/or the web server will check to see that the data is in the correct format and within the constraints set by the application. Validation done in the browser is called **client-side** validation, while validation done on the server is called **server-side** validation. In this chapter we are focusing on client-side validation.

If the information is correctly formatted, the application allows the data to be submitted to the server and (usually) saved in a database; if the information isn't correctly formatted, it gives the user an error message explaining what needs to be corrected, and lets them try again.

One of the most significant features of [HTML5 form controls](#) is the ability to validate most user data. This is done by using validation attributes on form elements. We've seen many of these earlier in the unit, but to recap:

- [required](#) : Specifies whether a form field needs to be filled in before the form can be submitted.
- [minlength](#) and [maxlength](#) : Specifies the minimum and maximum length of textual data (strings). You can constrain the character length of all text fields created by `<input>` or `<textarea>` using these attributes. A field is invalid if it has fewer characters than the `minlength` value or more than the `maxlength` value.
- [min](#) and [max](#) : Specifies the minimum and maximum values of numerical input types.
- [type](#) : Specifies whether the data needs to be a number, an email address, or some other specific preset type.
- [pattern](#) : Specifies a [regular expression](#) that defines a pattern the entered data needs to follow.

If the data entered in a form field follows all of the rules specified by the above attributes, it is considered valid. If not, it is considered invalid.

5.6.1.1. Further reading

We want to make filling out web forms as easy as possible. So why do we insist on validating our forms? There are three main reasons:

- **We want to get the right data, in the right format.** Our applications won't work properly if our users' data is stored in the wrong format, is incorrect, or is omitted altogether.
- **We want to protect our users' data.** Forcing our users to enter secure passwords makes it easier to protect their account information.
- **We want to protect ourselves.** There are many ways that malicious users can misuse unprotected forms to damage the application (see [Website security](#)).

Keeping this in mind, client-side validation *should not be considered* an exhaustive security measure! Your apps should always perform security checks on any form-submitted data on the *server-side* as well as the client-side, because client-side validation is too easy to bypass, so malicious users can still easily send bad data through to your server. Read [Website security](#) for an idea of what *could* happen; implementing server-side validation is somewhat beyond the scope of this module, but you should bear it in mind.

5.6.2. On the server side: retrieving the data

Whichever HTTP method you choose, the server receives a string that will be parsed in order to get the data as a list of key/value pairs. The way you access this list depends on the development platform you use and on any specific frameworks you may be using with it.

[PHP](#) offers some global objects to access the data. Assuming you've used the `GET` method, the example in the next sections just takes the data and saves it to a file. Of course, what you do with the data is up to you. You might display it, store it into a database, send it by email, or process it in some other way. We will use PHP to complete our examples.

5.6.3. The GET method

The `GET` method is the method used by the browser to ask the server to send back a given resource: "Hey server, I want to get this resource". In this case, the browser sends an empty body. Because the body is empty, if a form is sent using this method the data sent to the server is appended to the URL.

Considering our contact form, and keeping in mind that `GET` method has been used, when we submit the form, we'll see that the data appear in the URL at the browser address bar. For example, if

you enter “Fernando” as the user name, “fernando@test.com” as the email address, and “Hello” as the message, and you press the submit button, you should see something like this in the address bar: “ `contact.php?name=Fernando&email=fernando@test.com&message=Hello` ”.

The data is appended to the URL as a series of name/value pairs. After the URL web address has ended, a question mark is included (`?`) followed by the name/value pairs, each one separated by an ampersand (`&`). In this case we are passing three pieces of data to the server:

- `name` , which has a value of “Fernando”
- `email` , which has a value of “fernando@test.com”
- `message` , which has a value of “Hello”

5.6.3.1. Proposed exercise: Contact form

Create a new web page with a contact form, using the code in the previous example. It should look like the one below (probably not so nice). Check the result in your browser and validate your code. Also try to send the data by pressing the button and check the URL inside the address bar. Finally set the minimum and maximum length of the text fields to any values you consider suitable to ensure the data in this form is correct before sending it to the server.

— Note that if you press the submit button, you will go to the “`contact.php`” page, which is not implemented yet. At this point you will get an error, but you will see all the information in the URL, since we are using the GET method.

5.6.3.2. Proposed exercise: Full contact form

Let’s go ahead with some simple PHP code to save our data from the contact form. Create a file “`contact.php`” with the code below. Upload the form and the php code to your server and test your full example of the contact form to check that the messages are now saved into the server. Also tell some friends to test the web page and check that the data they have entered is also saved.

1. `<?php`

```

2.    // The global $_GET variable allows you to access the data sent with the GET
method by name
3.    $name = $_GET['name'];
4.    $email = $_GET['email'];
5.    $message = $_GET['message'];
6.
7.    // We put all data into the file "messages.csv" in a new line each time
8.    file_put_contents("messages.csv", "$name;$email;$message\n", FILE_APPEND);
9.
10.   // We show a link to the previous page and also to the file to check the
results
11.   echo "<p>Data saved</p>";
12.   echo "<p>Click <a href='". $_SERVER['HTTP_REFERER']."'>here</a> to go
back</p>";
13.   echo "<p>Click <a href='messages.csv' target='_blank'>here</a> to see all
messages</p>";
14.   ?>

```

5.6.3.3. Proposed exercise: Greetings form

Create a new web page with a form similar to the one below, check the result in your browser and validate the code. Press the submit button and have a look at the browser address bar. After that enter another data different from the default values, press the submit button and check that the new URL contains the right information. Finally change the default value of both text fields (“Hi” and “Mom”) to use some other values, and check the result again.

— Note that if you press the submit button, you will go to the “greetings.php” page, which is not implemented yet. At this point you will get an error, but you will see all the information in the URL, since we are using the GET method.

```

1.    <form action="greetings.php" method="GET">
2.        <p>
3.            <label>
4.                What greeting do you want to say?: <input name="say" value="Hi" required
/>
5.            </label>
6.        </p>
7.        <p>
8.            <label>

```

```
9.         Who do you want to say it to?: <textarea name="to"
required>Mom</textarea>
10.         </label>
11.     </p>
12.     <p>
13.         <button>Send my greetings</button>
14.     </p>
15. </form>
```

5.6.3.4. Proposed exercise: Full greetings form

Let's go ahead with some simple PHP code to save our data from the greetings form. Create a file "greetings.php" with the code below. Upload the form and the php code to your server and test your full example of the greetings form to check that the greetings are now saved into the server. Also tell some friends to test the web page and check that the data they have entered is also saved.

— You will see the similarities from the previous example (we have only changed the variables (`$say` and `$to`) and the file name where the data is saved ("greetings.csv").

```
1.  <?php
2.      // The global $_GET variable allows you to access the data sent with the GET
method by name
3.      $say = $_GET['say'];
4.      $to = $_GET['to'];
5.
6.      // We put all data into the file "greetings.csv" in a new line each time
7.      file_put_contents("greetings.csv", "$say,$to\n", FILE_APPEND);
8.
9.      // We show a link to the previous page and also to the file to check the
results
10.     echo "<p>Data saved</p>";
11.     echo "<p>Click <a href='".$_SERVER['HTTP_REFERER']."'>here</a> to go
back</p>";
12.     echo "<p>Click <a href='greetings.csv' target='_blank'>here</a> to see all
messages</p>";
13.  ?>
```

5.6.4. The POST method

The `POST` method is a little different. It's the method the browser uses to talk to the server when asking for a response that takes into account the data provided in the body of the HTTP request: "Hey server, take a look at this data and send me back an appropriate result". If a form is sent using this method, the data is appended to the body of the HTTP request instead of the URL. It is more secure than the `GET` method, since when the form is submitted using the `POST` method, the data cannot be seen by any other person around. This method is recommended for example to be used in forms where a password is sent.

Let's look at the following example, which is quite similar to the form in the `GET` section above, but with the `method` attribute set to `POST` and the `type` of the input box set to "password":

```
1. <form action="login.php" method="POST">
2.   <p>
3.     <label>User: <input type="text" name="user" required /></label>
4.   </p>
5.   <p>
6.     <label>Password: <input type="password" name="password" required /></label>
7.   </p>
8.   <p>
9.     <button>Check user and password</button>
10.  </p>
11. </form>
```

5.6.4.1. Proposed exercise: Login form

Create a new web page with a login form, using the code in the previous example. It should look like the one below (probably not so nice). Check the result in your browser and validate your code. Also try to send the data by pressing the button and check if there is any information in the URL. Finally set the minimum length of the user text field to 5 and the maximum to 10, and do the same for the password field.

— Note that if you press the submit button, you will go to the "login.php" page, which is not implemented yet. At this point you will get an error, but you will not see any information in the URL, since we are using the `POST` method.

5.6.4.2. Proposed exercise: Full login form

Let's go ahead with some simple PHP code to check the user and the password from the login form. Create a file "login.php" with the code below. Upload the form and the php code to your server and test your full example of the login form to check the user ("admin") and the password ("1234"). Also tell some friends to test your web page. After that, change the password from the file "login.php" and ask your friends to try to guess your new password. You must use a very simple password from "https://en.wikipedia.org/wiki/List_of_the_most_common_passwords" (otherwise your friends may be not able to guess it).

— You will see the similarities from the previous example (we have only changed the variables (`$user` and `$password`) and we have used a condition to show an image with a thumb up or down, depending on whether the password is correct or not.

```
1.  <?php
2.      // The global $_POST variable allows you to access the data sent with the
    POST method by name
3.      $user = $_POST['user'];
4.      $password = $_POST['password'];
5.
6.      // Check the user and the password
7.      if ($user == "admin" && $password == "1234") {
8.          echo "<img
    src='https://raw.githubusercontent.com/twbs/icons/main/icons/hand-thumbs-up.svg'
    width='100' />";
9.          echo "<p>Perfect! :-)</p><p>Click <a
    href='\"".$_SERVER['HTTP_REFERER']."'>here</a> to go back</p>";
10.     }
11.     else {
12.         echo "<img
    src='https://raw.githubusercontent.com/twbs/icons/main/icons/hand-thumbs-
    down.svg' width='100' />";
13.         echo "<p>Invalid user or password! :-(</p><p>Click <a
    href='\"".$_SERVER['HTTP_REFERER']."'>here</a> to try again</p>";
14.     }
15.     ?>
```