Chapter 4. Web Forms, Refined

*HTML forms* are simple HTML controls you use to collect information from website visitors. They include text boxes people can type into, list boxes they can pick from, checkboxes they can switch on or off, and so on. There are countless ways to use HTML forms, and if you’ve kicked around the Web for more than a week, you’ve used them to do everything from getting a stock quote to signing up for an email account.

HTML forms have existed almost since the dawn of HTML, and they haven’t changed a wink since last century, despite some serious efforts. Web standards-makers spent years cooking up a successor called XForms, which fell as flat as XHTML 2 (see [XHTML 1.0: Getting Strict](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch01.html#xhtml_10_getting_strict)). Although XForms solved some problems easily and elegantly, it also had its own headaches—for example, XForms code was verbose and assumed that web designers were intimately familiar with XML. But the biggest hurdle was the fact that XForms wasn’t compatible with HTML forms in any way, meaning that developers would need to close their eyes and jump to a new model with nothing but a whole lot of nerve and hope. But because mainstream web browsers never bothered to implement XForms—it was too complex and little used—web developers never ended up taking that leap.

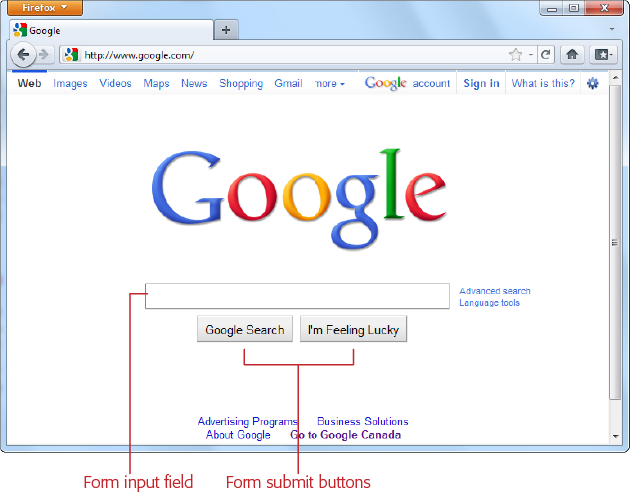
HTML5 takes a different approach. It introduces refinements to the existing HTML forms model, which means HTML5-enhanced forms can keep working on older browsers, just without all the bells and whistles. (This is good, because Internet Explorer won’t start supporting the new form features until IE 10.) HTML5 forms also focus on adding features that developers are already using today. The difference is that HTML5 makes them easily accessible, without requiring a tangle of JavaScript code or a JavaScript toolkit from another company.

In this chapter, you’ll tour all the new features of HTML5 forms. You’ll see which ones are supported, which ones aren’t, and which workarounds can help you smooth over the differences. You’ll also consider a feature that isn’t technically part of the HTML5 forms standard but is all about interactivity—putting a rich HTML editor in an ordinary web page.

Understanding Forms

Odds are that you’ve worked with forms before. But if you’re a bit sketchy on the details, the following recap will refresh your memory.

A *web form* is a collection of text boxes, lists, buttons, and other clickable widgets that a web surfer uses to supply some sort of information to a website. Forms are all over the Web—they allow you to sign up for email accounts, review products, and make bank transactions. The simplest possible form is the single text box that adorns search engines like Google (see [Figure 4-1](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#googles_spartan_search_page_holds_a_basi)).



*Figure 4-1. Google’s Spartan search page holds a basic HTML form. You use it in the same way you use any form—enter some information (in this case, search keywords), and click a button to submit that information.*

All basic web forms work in the same way. The user fills in some information and then clicks a button. At that point, the server collects all the data that the user has entered and sends it back to the web server. On the web server, some sort of application digests the information and takes the appropriate next step. The server-side program might consult a database (either to read or to store some information), before sending a new page back to the web browser.

The tricky part of this discussion is that there are hundreds of ways to build the server-side part of the equation (that’s the application that processes the information that’s submitted from the form). Some developers may use stripped-down scripts that let them manipulate the raw form data, while others may work with higher-level models that package the form details in neat programming objects. But either way, the task is basically the same—examine the form data, do something with it, and then send back a new page.

NOTE

This book doesn’t make any assumptions about your choice of server-side programming tool. In fact, it doesn’t really matter, because you still need to use the same set of form elements, and these elements are still bound by the same HTML5 rules.

**UP TO SPEED: BYPASSING FORM SUBMISSION WITH JAVASCRIPT**

It’s worth noting that forms aren’t the only way to send user-entered information to a web server (although they were, once upon a time). Today, crafty developers can use the XMLHttpRequest object ([The XMLHttpRequest Object](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch11.html#the_xmlhttprequest_object)) in JavaScript code to quietly communicate with a web server. For example, the Google search page uses this approach in two different ways—first, to get search suggestions, which it displays in a drop-down list; and second, to get a search results page as you type, if you’ve enabled the Google Instant feature ([www.google.com/instant](http://www.google.com/instant)).

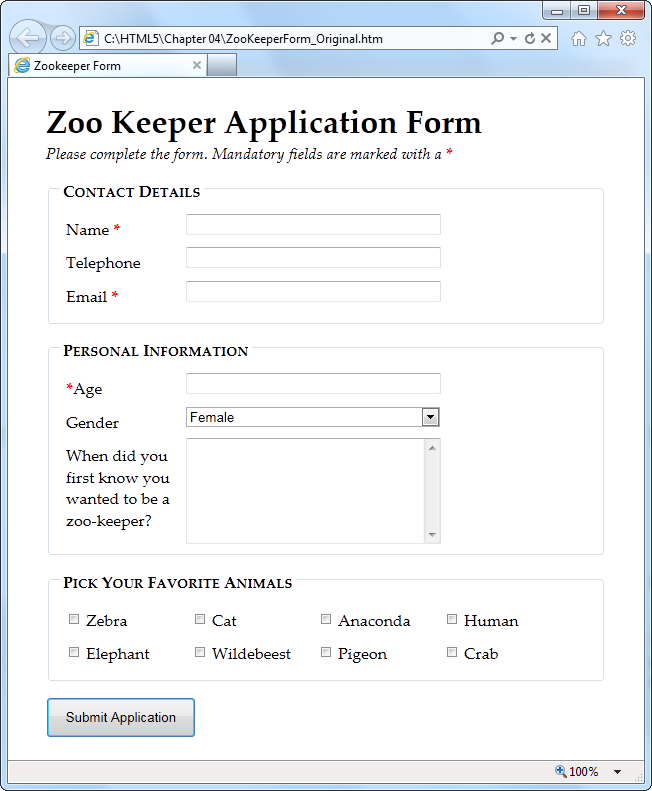
It might occur to you that JavaScript can completely bypass the form submission step, as it does in Google Instant. But while it’s possible to offer this as a *feature*, it’s not acceptable to include it as a *requirement*. That’s because the JavaScript approach isn’t bulletproof (for example, it may exhibit the occasional quirk on a slow connection) and there’s still a small fraction of people with no JavaScript support or with JavaScript turned off in their browsers.

Finally, it’s worth noting that it’s perfectly acceptable to have a page that never submits its form. You’ve probably seen pages that perform simple calculations (for example, a mortgage interest rate calculator). These forms don’t need any help from the server, so they can perform their calculations entirely in JavaScript and display the result on the current page.

From the HTML5 point of view, it really doesn’t matter whether you submit your form to a server, use the data in an ordinary JavaScript routine, or pass it back to the server through XMLHttpRequest. In all cases, you’ll still build your form using the standard HTML forms controls.

Revamping a Traditional HTML Form

The best way to learn about HTML5 forms is to take a typical example from today and enhance it. [Figure 4-2](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#if_youve_traveled_the_web_youve_seen_you) shows the example you’ll start out with.



*Figure 4-2. If you’ve traveled the Web, you’ve seen your share of forms like this one, which collects basic information from a web page visitor.*

The markup is dishwater-dull. If you’ve worked with forms before, you won’t see anything new here. First, the entire form is wrapped in a <form> element:

**<form** id="zooKeeperForm" action="processApplication.cgi"**>**

**<p><i>**Please complete the form. Mandatory fields are marked with

a **</i><em>**\***</em></p>**

...

The <form> element bundles together all the form widgets (also known as *controls* or *fields*). It also tells the browser where to post the page when it’s submitted, by providing a URL in the action attribute. If you plan to do all the work in client-side JavaScript code, you can simply use a number sign (#) for the action attribute.

NOTE

HTML5 adds a mechanism for placing form controls outside of the form to which they belong. The trick is to use the new form attribute to refer to the form by its id value (as in*form=“zooForm”*). However, browsers that don’t support this feature will completely overlook your data when the form is submitted, which means this minor feature is still too risky to use in a real web page.

A well-designed form, like the zookeeper application, divides itself into logical chunks using the <fieldset> element. Each chunk gets a title, courtesy of the <legend> element. Here’s the <fieldset> for the Contact Details section (which is dissected in [Figure 4-3](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#this_ltfieldsetgt_collects_three_pieces_)):

...

**<fieldset>**

**<legend>**Contact Details**</legend>**

**<label** for="name"**>**Name **<em>**\***</em></label>**

**<input** id="name"**><br>**

**<label** for="telephone"**>**Telephone**</label>**

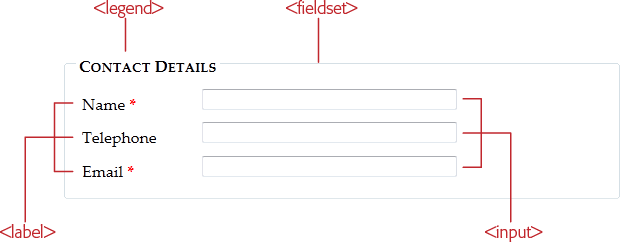
**<input** id="telephone"**><br>**

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

**<input** id="email"**><br>**

**</fieldset>**

...



*Figure 4-3. This <fieldset> collects three pieces of information: a name, telephone number, and email address. For each piece of information, you supply a caption (using the <label> element) and a control to collect the data (using an <input>, <textarea>, or <select> element).*

As in all forms, the bulk of the work is handled by the all-purpose <input> element, which collects text and creates checkboxes, radio buttons, and list buttons. Along with <input>, the <textarea> element gives people a way to enter multiple lines of text, and the <select> element creates a list. If you need a refresher, [Table 4-1](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#form_controls) will fill you in.

*Table 4-1. Form controls*

| **Control** | **HTML element** | **Description** |
| --- | --- | --- |
| Single-line textbox | **<input** type="text"**>**  **<input** type="password"**>** | Shows a text box where visitors can type in text. If you use the password type, the browser won’t display the text. Instead, visitors see an asterisk (\*) or a bullet (•) in place of each letter as they type in their password. |
| Multiline textbox | **<textarea>**...**</textarea>** | Shows a large text box that can fit multiple lines of text. |
| Checkbox | **<input** type="checkbox"**>** | Shows a checkbox that can be switched on or off. |
| Radio button | **<input** type="radio"**>** | Shows a radio button (a circle you can turn on or off). Usually, you have a group of radio buttons with the same value for the *name* attribute, in which case the visitor can select only one. |
| Button | **<input** type="submit"**>**  **<input** type="image"**>**  **<input** type="reset"**>**  **<input** type="button"**>** | Shows the standard clickable button. A *submit* button always gathers up the form data and sends it to its destination. An*image* button does the same thing, but it lets you display a clickable picture instead of the standard text-on-a-button. A *reset* button clears the visitor’s selections and text from all the input controls. A *button* button doesn’t do anything unless you add some JavaScript code. |
| List | **<select>**...**</select>** | Shows a list where your visitor can select one or more items. You add an <option> element for each item in the list. |

Here’s the rest of the zookeeper form markup, with a few new details (a <select> list, checkboxes, and the button that submits the form):

...

**<fieldset>**

**<legend>**Personal Information**</legend>**

**<label** for="age"**><em>**\***</em>**Age**</label>**

**<input** id="age"**><br>**

**<label** for="gender"**>**Gender**</label>**

**<select** id="gender"**>**

**<option** value="female"**>**Female**</option>**

**<option** value="male"**>**Male**</option>**

**</select><br>**

**<label** for="comments"**>**When did you first know you wanted to be a

zoo-keeper?**</label>**

**<textarea** id="comments"**></textarea>**

**</fieldset>**

**<fieldset>**

**<legend>**Pick Your Favorite Animals**</legend>**

**<label** for="zebra"**><input** id="zebra" type="checkbox"**>** Zebra**</label>**

**<label** for="cat"**><input** id="cat" type="checkbox"**>** Cat**</label>**

**<label** for="anaconda"**><input** id="anaconda" type="checkbox"**>** Anaconda

**</label>**

**<label** for="human"**><input** id="human" type="checkbox"**>** Human**</label>**

**<label** for="elephant"**><input** id="elephant" type="checkbox"**>** Elephant

**</label>**

**<label** for="wildebeest"**><input** id="wildebeest" type="checkbox"**>**

Wildebeest**</label>**

**<label** for="pigeon"**><input** id="pigeon" type="checkbox"**>** Pigeon**</label>**

**<label** for="crab"**><input** id="crab" type="checkbox"**>** Crab**</label>**

**</fieldset>**

**<p><input** type="submit" value="Submit Application"**></p>**

**</form>**

You can find the full example, along with the relatively simple style sheet that formats it, on the try-out site ([www.prosetech.com/html5](http://www.prosetech.com/html5)). Look for the ZookeperForm\_Original.html file to play around with traditional, unenhanced version of the form, and ZookeperForm\_Revised.html to get all the HTML5 goodies.

NOTE

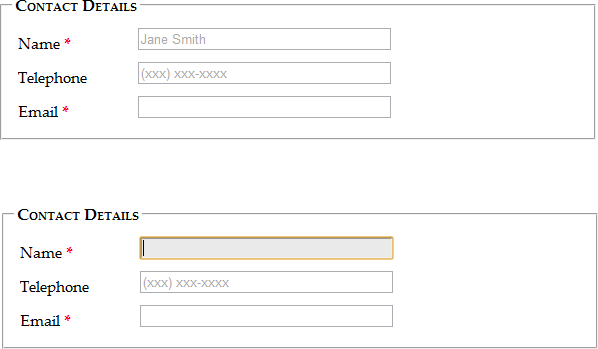
One limit of HTML forms is that you can’t change how the browser draws controls. For example, if you want to replace the standard dull gray checkbox with a big black-and-white box with a fat red checkmark image, you can’t. (The alternative is to create a normal element that has checkbox-like behavior using JavaScript—in other words, it changes its appearance back and forth when someone clicks it.)

HTML5 keeps this no-customization limit in place, and extends it to the new controls you’ll learn about in this chapter. That means forms won’t suit people who need highly stylized widgets and complete control over the look of their pages.

Now that you’ve got a form to work with, it’s time to start improving it with HTML5. In the following sections, you’ll start small, with placeholder text and an autofocus field.

Adding Hints with Placeholders

Forms usually start out empty. But a column of blank text boxes can be a bit intimidating, especially if it’s not absolutely clear what belongs inside each text box. That’s why you commonly see some sort of sample text inside otherwise-empty text boxes. This placeholder text is also called a *watermark*, because it’s often given a light gray color to distinguish it from real, typed-in content. [Figure 4-4](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#top_when_a_field_is_empty_its_placeholde) shows a placeholder in action.



*Figure 4-4. Top: When a field is empty, its placeholder text appears, as with the Name and Telephone fields shown here. Bottom: When the user clicks in the field (giving it focus), the placeholder text disappears. When the form filler moves on to another field, the placeholder text reappears, as long as the text box is still empty.*

To create a placeholder, simply use the *placeholder* attribute:

**<label** for="name"**>**Name **<em>**\***</em></label>**

**<input** id="name" **placeholder="Jane Smith"**>**<br>**

**<label** for="telephone"**>**Telephone**</label>**

**<input** id="telephone" **placeholder="(xxx) xxx-xxxx"**>**<br>**

Browsers that don’t support placeholder text just ignore the placeholder attribute; Internet Explorer is the main culprit. Fortunately, it’s not a big deal, since placeholders are just nice form frills, not essential to your form’s functioning. If it really bothers you, there are plenty of JavaScript patches that can bring IE up to speed, painlessly, at <http://tinyurl.com/polyfills>.

Right now, there’s no standard, consistent way to change the appearance of placeholder text (for example, to italicize it or to change the text color). Eventually, browser makers will create the CSS styling hooks that you need—in fact, they’re hashing out the details even as you read this. But for now, you’ll either need to fiddle with browser-specific pseudoclasses (namely, *-webkit-input-placeholder* and *-mozplaceholder*) or leave it alone. ([Pseudoclass Selectors](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/apa.html#pseudoclass_selectors) explains pseudoclasses.)

However, you can use the better-supported *focus* pseudoclass to change the way a text box looks when it gets the focus. For example, you might want to assign a darker background color to make it stand out:

**input**:focus {

**background**: #eaeaea;

}

**UP TO SPEED: WRITING GOOD PLACEHOLDERS**

You don’t need placeholders for every text box. Instead, you should use them to clear up potential ambiguity. For example, no one needs an explanation about what goes in a First Name box, but the Name box in [Figure 4-4](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#top_when_a_field_is_empty_its_placeholde) isn’t quite as obvious. The placeholder text makes it clear that there’s room for a first and last name.

Sometimes placeholders include a sample value—in other words, something you might actually type into the box. For example, Google’s recipe search (<http://www.google.com/landing/recipes>) uses the text *chicken pasta* for a placeholder, making it obvious that you should enter part of a recipe name, not a list of ingredients or the name of the chef who cooked it up.

Other times, placeholders indicate the way a value should be formatted. The telephone box in [Figure 4-4](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#top_when_a_field_is_empty_its_placeholde) is an example—it shows the value *(xxx) xxx-xxxx* to concisely indicate that telephone numbers should consist of a three-digit area code, followed by a sequence of three, then four digits. This placeholder doesn’t necessarily mean that differently formatted input isn’t allowed, but it does offer a suggestion that uncertain people can follow.

There are two things you shouldn’t try with a placeholder. First, don’t try to cram in a description of the field or instructions. For example, imagine you have a box for a credit card’s security code. The text, “The three digits listed on the back of your card” is *not* a good placeholder. Instead, consider adding a text note under the input box, or using the title attribute to make a pop-up window appear when someone hovers over the field:

**<label** for="promoCode"**>**Promotion Code**</label>**

**<input** id="promoCode" placeholder="QRB001"

**title="Your promotion code is three letters**

**followed by three numbers">**

Second, you shouldn’t add special characters to your placeholder in an attempt to make it obvious that your placeholder is not real, typed-in text. For example, some websites use placeholders like *[John Smith]* instead of *John Smith*, with the square brackets there to emphasize that the placeholder is just an example. This convention can be confusing.

Focus: Starting in the Right Spot

After loading up your form, the first thing your visitors want to do is start typing. Unfortunately, they can’t—at least not until they tab over to the first control, or click it with the mouse, thereby giving it *focus*.

You can make this happen with JavaScript, by calling the focus() method of the appropriate <input> element. But this involves an extra line of code and can sometimes cause annoying quirks. For example, it’s sometimes possible for the user to click somewhere else and start typing before the focus() method gets called, at which point focus is rudely transferred back to the first control. But if the browser were able to control the focus, it could be a bit smarter, and transfer focus only if the user hasn’t already dived into another control.

That’s the idea behind HTML5’s new *autofocus* attribute, which you can add to a single <input> or <textarea> element, like this:

**<label** for="name"**>**Name **<em>**\***</em></label>**

**<input** id="name" placeholder="Jane Smith" **autofocus**>**<br>**

The autofocus has similar support as the placeholder attribute, which means basically every browser recognizes it except Internet Explorer. Once again, it’s easy enough to plug the hole. You can check for autofocus support using Modernizr ([Feature Detection with Modernizr](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch01.html#feature_detection_with_modernizr)) and then run your own autofocus code if needed. Or, you can use a ready-made JavaScript polyfill that adds autofocus support (<http://tinyurl.com/polyfills>). However, it hardly seems worth it for such a minor frill, unless you’re also aiming to give IE support for other form features, like the validation system discussed next.

Validation: Stopping Errors

The fields in a form are there to gather information from web page visitors. But no matter how politely you ask, you might not get what you want. Impatient or confused visitors can skip over important sections, enter partial information, or just hit the wrong keys. The end result? They click Submit, and your website gets a whackload of scrambled data.

What a respectable web page needs is *validation*—a way to catch mistakes when they happen (or even better, to prevent them from happening at all). For years, developers have done that by writing their own JavaScript routines or using professional JavaScript libraries. And, truthfully, these approaches work perfectly well. But seeing as validation is so common (just about everyone needs to do error-checking), and seeing as validation generally revolves around a few key themes (for example, spotting invalid email addresses or dates), and seeing as validation is boring (no one really wants to write the same code for every form, not to mention *test* it), there’s clearly room for a better way.

The creators of HTML5 spotted this low-hanging fruit and invented a way for browsers to help out, by getting them to do the validation work instead of web developers. They devised a *client-side* validation system (see the box on[Validating in Two Places](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#validating_in_two_places)) that lets you embed common error-checking rules into any <input> field. Best of all, this system is easy—all you need to do is insert the right attribute.

How HTML5 Validation Works

The basic idea behind HTML5 form validation is that you indicate where validation should happen, but you don’t actually *implement* the tedious details. It’s a bit like being promoted into a management job, just without the pay raise.

For example, imagine you decide a certain field cannot be left blank—the form filler needs to supply some sort of information. In HTML5, you can make this demand by adding the *required* attribute:

**<label** for="name"**>**Name **<em>**\***</em></label>**

**<input** id="name" placeholder="Jane Smith" autofocus **required**>**<br>**

**UP TO SPEED: VALIDATING IN TWO PLACES**

Throughout the years, crafty developers have attacked the validation problem in different ways. Today, the consensus is clear. To make a bulletproof form, you need two types of error-checking:

* **Client-side validation**. These are the checks that happen in the browser, *before* a form is submitted. The goal here is to make life easier for the people filling out the form. Instead of waiting for them to complete three dozen text boxes and click a submit button, you want to catch problems in the making. That way you can pop up a helpful error message right away and in the right spot, allowing the form filler to correct the mistake before submitting the form to the server.
* **Server-side validation**. These are the checks that happen *after* a form is sent back to the web server. At this point, it’s up to your server-side code to review the details and make sure everything is kosher before continuing. No matter what the browser does, server-side validation is essential. It’s the only way to defend yourself from malicious people who are deliberately trying to tamper with form data. If your server-side validation detects a problem, you send back a page with an error message.

So client-side validation (of which HTML5 validation is an example) is there to make life easier for your web page visitors, while server-side validation ensures correctness. The key thing to understand is that you need both types of validation—unless you have an exceedingly simple form where mistakes aren’t likely or aren’t a big deal.

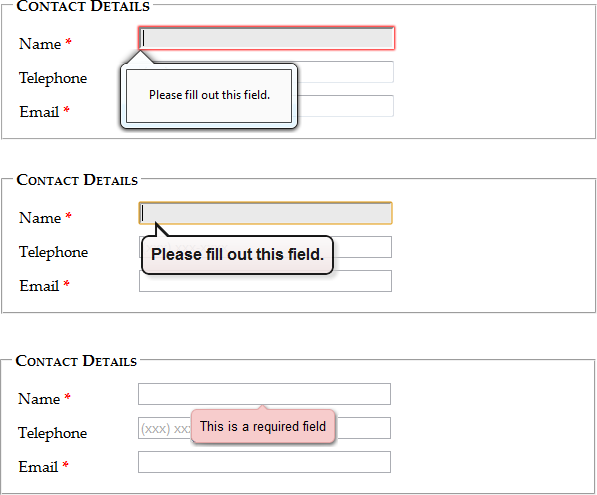
Initially, there’s no visual detail to indicate that a field is required. For that reason, you might want to use some other visual clue, such as giving the text box a different border color or placing an asterisk next to the field (as in the zookeeper form).

Validation kicks in only when the form filler clicks a button to submit the form. If the browser implements HTML5 forms, then it will notice that a required field is blank, intercept the form submission attempt, and show a pop-up message that flags the invalid field ([Figure 4-5](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#heres_the_same_required_field_in_firefox)).

As you’ll see in the following sections, different attributes let you apply different error-checking rules. You can apply more than one rule to the same input box, and you can apply the same rule to as many <input> elements as you want (and to the <textarea> element). All the validation conditions must be met before the form can be submitted.

This raises a good question: What happens if form data breaks more than one rule—for example, it has multiple required fields that aren’t filled in?

Once again, nothing happens until the person filling out the form clicks the submit button. Then, the browser begins examining the fields from top to bottom. When it finds the first invalid value, it stops checking any further. It cancels the form submission and pops up an error message next to this value. (Additionally, if the offending text box isn’t currently visible, the browser scrolls up just enough that it appears at the top of the page.) If the visitor corrects the problem and clicks the submit button again, the browser will stop and highlight the next invalid value.



*Figure 4-5. Here’s the same required field in Firefox (top), Chrome (middle), and Opera (bottom). Browsers are free to choose the exact way they notify people about validation problems, but they all use a pop-up box that looks like a stylized tooltip. Unfortunately, you can’t customize the formatting of this box or change the wording of the validation message—at least not yet.*

NOTE

Web browsers hold off on validation until a submit button is clicked. This ensures that the validation system is efficient and restrained, so it works for everyone.

Some web developers prefer to alert people as soon as they leave an invalid field (by tabbing away or clicking somewhere else with the mouse). This sort of validation is handy in long forms, especially if there’s a chance that someone may make a similar mistake in several different fields. Unfortunately, HTML5 doesn’t have a way for you to dictate when the web browser does its validation, although it’s possible that it might add one in the future. For now, if you want immediate validation messages, it’s best to write the JavaScript yourself or to use a good JavaScript library.

Turning Validation Off

In some cases, you may need to disable the validation feature. For example, you might need to turn it off for testing to verify that your server-side code deals appropriately with invalid data. To turn validation off for an entire form, you add the *novalidate* attribute to the containing <form> element:

**<form** id="zooKeeperForm" action="processApplication.cgi" **novalidate**>

The other option is to provide a submit button that bypasses validation. This technique is sometimes useful in a web page. For example, you may want to enforce strict validation for the official submit button, but provide another button that does something else (like storing half-completed data for later use). To allow this, add the *formnovalidate* attribute to the <input> element that represents your button:

**<input** type="submit" value="Save for Later" **formnovalidate**>

You’ve now seen how to use validation to catch missing information. Next, you’ll learn to search for errors in different types of data.

NOTE

Planning to validate numbers? There’s no validation rule that forces text to contain digits, but there is a new *number* data type, which you’ll examine on [Telephone Numbers](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#telephone_numbers). Unfortunately, its support is still sketchy.

Validation Styling Hooks

Although you can’t style validation messages, you can change the appearance of the input fields based on their validation *state*. For example, you can give invalid values a different background color, which will appear in the text box as soon as the browser detects the problem.

To use this technique, you simply need to add a few new pseudoclasses ([Pseudoclass Selectors](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/apa.html#pseudoclass_selectors)). Your options include:

* **required** and **optional**, which apply styles to fields based on whether they use the required attribute.
* **valid** and **invalid**, which apply styles to controls based on whether they contain mistakes. But remember that most browsers won’t actually discover invalid values until the visitor tries to submit the form, so you won’t see the invalid formatting right away.
* **in-range** and **out-of-range**, which apply formatting to controls that use the min and max attributes to limit numbers to a range ([Telephone Numbers](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#telephone_numbers)).

For example, if you want to give required <input> fields a light yellow background, you could use a style rule with the *required* pseudoclass:

**input**:required {

**background-color**: lightyellow;

}

Or, you might want to highlight only those fields that are required and currently hold invalid values by combining the *required* and *invalid*pseudoclasses like this:

**input**:required:invalid {

**background-color**: lightyellow;

}

With this setting, blank fields are automatically highlighted, because they break the required-field rule.

You can use all sorts of other tricks, like combining the validation pseudoclasses with the *focus* pseudoclass, or using an offset background that includes an error icon to flag invalid values. Of course, a hefty disclaimer applies: You can use these pseudoclasses to improve your pages, but make sure your form still looks good without them, because support lags in older browsers.

Validating with Regular Expressions

The most powerful (and complex) type of validation that HTML5 supports is based on regular expressions. Seeing as JavaScript already supports regular expression, adding this feature to HTML forms makes perfect sense.

A *regular expression* is a pattern written using the regular expression language. Regular expressions are designed to match patterned text—for example, a regular expression can make sure that a postal code has the right sequence of letters and digits, or that an email address has an @ symbol and a domain extension that’s at least two characters long. For example, consider this expression:

[A-Z]{3}-[0-9]{3}

The square brackets at the beginning define a range of allowed characters. In other words, [A-Z] allows any uppercase letter from A to Z. The curly brackets that follow multiply this effect, so {3} means you need three uppercase letters. The dash that follows doesn’t have a special meaning, so it indicates that a dash must follow the three-letter sequence. Finally, [0-9] allows a digit from 0 to 9, and {3} requires three of them.

Regular expression matching is useful for searching (finding pattern matches in a long document) and validation (verifying that a value matches a pattern). HTML5 forms use regular expressions for validation.

NOTE

Regular expression geeks take note: You don’t need the magic ^ and $ characters to match the beginning or end of a value in a field. HTML5 assumes both details automatically, which means a regular expression must match the *entire* value in a field in order to be deemed valid.

These values are valid, because they match the pattern shown above:

QRB-001

TTT-952

LAA-000

But these values are not:

qrb-001

TTT-0952

LA5-000

Regular expressions quickly get much more complex than this example. Writing a regular expression can be quite a chore, which is why most developers simply search for a ready-made regular expression that validates the type of data they want to check. Or they get help.

TIP

To learn just enough about the regular expression language to make your own super-simple expressions, check out the concise tutorials at [www.w3schools.com/js/js\_obj\_regexp.asp](http://www.w3schools.com/js/js_obj_regexp.asp) or <http://tinyurl.com/jsregex>. To find ready-made regular expressions that you can use with your forms, visit [http://regexlib.com](http://regexlib.com/). And to become a regular expression guru, read *Mastering Regular Expressions, Third Edition*by Jeffrey Friedl (O’Reilly).

Once you have a regular expression, you can enforce it in any <input> or <textarea> element by adding the *pattern* attribute:

**<label** for="promoCode"**>**Promotion Code**</label>**

**<input** id="promoCode" placeholder="QRB-001" title=

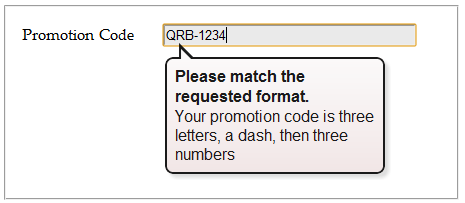
"Your promotion code is three uppercase letters, a dash, then three numbers"

**pattern="[A-Z]{3}-[0-9]{3}"**>

[Figure 4-6](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#smart_browsers_like_google_chrome_shown_) shows what happens if you break the regular expression rule.

TIP

Browsers don’t validate blank values. In this example, a blank promotion code passes muster. If this isn’t what you want, then combine the pattern attribute with the required attribute.



*Figure 4-6. Smart browsers (like Google Chrome, shown here) don’t just catch the mistake, they also grab the text from the title attribute and display that too to help guide the person who’s filling out the form.*

NOTE

Regular expressions seem like a perfect match for email addresses (and they are). However, hold off on using them this way, because HTML5 already has a dedicated input type for email addresses that has the right regular expression baked in ([Email Addresses](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#email_addresses)).

Custom Validation

The HTML5 specification also outlines a set of JavaScript properties that let you find out if fields are valid (or force the browser to validate them). The most useful of these is the setCustomValidity() method, which lets you write custom validation logic for specific fields and have it work with the HTML5 validation system.

Here’s how it works. First, you need to check the appropriate field for errors. You do this by handling the onInput event, which is nothing new:

**<label** for="comments"**>**When did you first know you wanted to be a

zookeeper?**</label>**

**<textarea** id="comments" **oninput="validateComments(this)"** >**</textarea>**

In this example, the onInput event triggers a function named validateComments(). It’s up to you to write this function, check the current value of the <input> element, and then call setCustomValidity().

If the current value has problems, you need to supply an error message when you call setCustomValidity(). Or, if the current value checks out, you need to call setCustomValidity() with an empty string. This clears any error custom messages that you may set earlier.

Here’s an example that forces the text in the comment box to be at least 20 characters long:

**function** validateComments(input) {

**if** (input.value.length < 20) {

input.setCustomValidity("You need to comment in more detail.");

}

**else** {

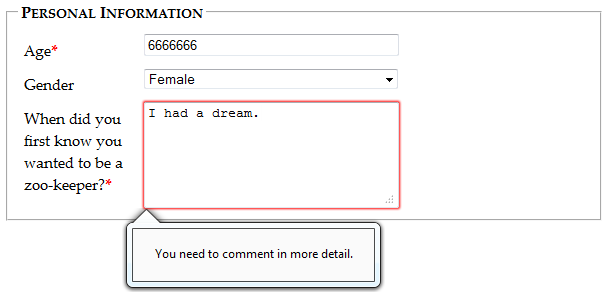
*// There's no error. Clear any error message.*

input.setCustomValidity("");

}

}

[Figure 4-7](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#when_you_supply_an_error_message_with_se) shows what happens if someone breaks this rule and then tries to submit the form.



*Figure 4-7. When you supply an error message with setCustomValidity(), the browser treats it the same as its own, built-in validation messages. Try to submit the form, and you’ll see a pop-up warning with your message text.*

Of course, you could solve this problem more neatly with a regular expression that requires long strings. But while regular expressions are great for validating some data types, custom validation logic can do *anything*, from complex algebra to contacting the web server.

NOTE

Remember, your web page visitors can see anything you put in JavaScript, so it’s no place for secret algorithms. For example, you might know that in a valid promotional code, the digits always add up to 12. But you probably don’t want to betray that detail in a custom validation routine, because it will help shifty people cook up fake codes. Instead, keep this sort of validation in the web server.

Browser Support for Validation

Browser makers added support for validation in pieces. That means some browser builds support some validation features while ignoring others. [Table 4-2](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#browser_support_for_validation-id001) indicates the minimum browser versions you need to use to get solid support for all the validation tricks you’ve learned so far.

*Table 4-2. Browser support for validation*

|  | **IE** | **Firefox** | **Chrome** | **Safari** | **Opera** | **Safari iOS** | **Android** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Minimum version | 10[[a](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#ftn.CHP-4-FNOTE-1)] | 4 | 10 | 5 (Windows only) | 10 | - | - |
| [[a](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#CHP-4-FNOTE-1)]*Currently, this version is available in early beta builds only*. | | | | | | | |

Because HTML5 validation doesn’t replace the validation you do on your web server, you may see it as a frill, in which case you can accept this uneven support. Browsers that don’t implement validation, like IE 9, let people submit forms with invalid dates, but you can then catch these problems on the web server and return the same page, but with error details.

On the other hand, your website might include complex forms with lots of potential for confusion, and you might not be ready to accept a world of frustrated IE users. In this case, you have two basic choices: Fall back on your own validation system, or use a JavaScript library that adds the missing smarts. Your choice depends on the extent and complexity of your validation.

If all your form needs is a smattering of simple validation, it’s probably worth adding your own checks. Using Modernizr ([Feature Detection with Modernizr](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch01.html#feature_detection_with_modernizr)), you can check for a variety of HTML5 web forms features. For example, use the Modernizr.input.pattern property to check whether the current browser recognizes the pattern attribute:

**if** (!Modernizr.input.pattern) {

*// The current browser doesn't perform regular expression validation.*

*// You can use the regular expression features in JavaScript instead.*

...

}

NOTE

The pattern property is just one of the properties exposed by Modernizr.input object. Other properties that are useful for testing web form support include placeholder, autofocus, required, max, min, and step.

Of course, this example doesn’t tell you *when* to perform this check or *how* to react. If you want your validation to mimic the HTML5 validation system, it makes sense to perform your validation when the person viewing the form attempts to submit it. You do this by handling the form’s onSubmit event, and then returning either true (which means the form is valid and the browser can submit it) or false (which means the form has a problem and the browser should cancel the operation):

**<form** id="zooKeeperForm" action="processApplication.cgi"

**onsubmit="return validateForm()">**

Here’s an example of a very simple custom validation routine that enforces required fields:

**function** validateForm() {

**if** (!Modernizr.input.required) {

*// The required attribute is not supported, so you need to check the*

*// required fields yourself.*

*// First, get an array that holds all the elements.*

**var** inputElements = document.getElementById("zooKeeperForm").elements;

*// Next, move through that array, checking each element.*

**for**(**var** i = 0; i < inputElements.length; i++) {

*// Check if this element is required.*

**if** (inputElements[i].hasAttribute("required")) {

*// If this element is required, check if it has a value.*

*// If not, the form fails validation, and this function returns false.*

**if** (inputElements[i].value == "") **return** **false**;

}

}

*// If you reach this point, everything worked out and the browser*

*// can submit the form.*

**return** **true**;

}

}

TIP

This block of code relies on a number of basic JavaScript techniques, including element lookup, a loop, and conditional logic. To learn more about all these details, check out [Appendix B](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/apb.html).

If you have a complex form and you want to save some effort (while at the same time preparing for the future), you may prefer to use a JavaScript patch to solve all your problems. Technically, the approach is the same—your page will check for validation support and then perform validation manually if necessary. The difference is that the JavaScript library already has all the tedious code you need.

At <http://tinyurl.com/polyfills>, you can find a long, intimidating list of JavaScript libraries that all attempt to do more or less the same thing. One of the best ones is simply called webforms2, and is available from <https://github.com/westonruter/webforms2> (to get a copy, look for the easy-to-miss Download button).

The webforms2 library implements all the attributes you’ve seen so far. To make it work, you download all the files to your website folder (or, more commonly, a subfolder of your website folder), and add a single reference to the library in your web page:

**<head>**

**<title>**...**</title>**

**<script** src="webforms2.js"**></script>**

...

**<head>**

The webforms2 library integrates nicely with another JavaScript patch, called html5Widgets, which adds support for the form features you’ll learn about next, like the slider, date picker, and color chooser. To learn more about this all-in-one patch, visit [www.useragentman.com/tests/html5Widgets](http://www.useragentman.com/tests/html5Widgets). Both of these libraries offer good all-around web forms support, but there are inevitable gaps and minor bugs buried in the code. Only time will tell if these already-impressive libraries are properly maintained and gradually improved.

**GEM IN THE ROUGH: A FEW ROGUE INPUT ATTRIBUTES**

HTML5 recognizes a few more attributes that can control browser behavior when editing forms, but aren’t used for validation. Not all of these attributes apply to all browsers. Still, they make for good experimenting:

* **Spellcheck**. Some browsers try to help you avoid embarrassment by spell-checking the words you type in an input box. The obvious problem is that not all text is meant to be real words, and there are only so many red squiggles a web surfer can take before getting just a bit annoyed. Set spellcheck to false to recommend that the browser not spellcheck a field, or true to recommend that it does. (Browsers differ on their default spell-checking behavior, which is what you get if you don’t set the spellcheck attribute at all.)
* **Autocomplete**. Some browsers try to save you time by offering you recently typed-in values when you enter information in a field. This behavior isn’t always appropriate—as the HTML5 specification points out, some types of information may be sensitive (like nuclear attack codes) or may be relevant for only a short amount of time (like a one-time bank login code). In these cases, set autocomplete to off to recommend that the browser not offer autocomplete suggestions. You can also set autocomplete to on to recommend it for a particular field.
* **Autocorrect** and **autocapitalize**. Use these attributes to control automatic correction and capitalization features on some mobile devices—namely, the version of Safari that runs on iPads and iPhones.
* **Multiple**. Web designers have been adding the multiple attribute to the <select> element to create multiple-selection lists since time immemorial. But now you can add it to certain types of <input> elements, including ones that use the file type (for uploading files) and ones that use the email type ([Email Addresses](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#email_addresses)). On a supporting browser, the user can then pick several files to upload at once, or stick multiple email addresses in one box.

New Types of Input

One of the quirks of HTML forms is that one ingredient—the vaguely named <input> element—is used to create a variety of controls, from checkboxes to text boxes to buttons. The *type* attribute is the master switch that determines what each <input> element really is.

If a browser runs into an <input> element with a type that it doesn’t recognize, the browser treats it like an ordinary text box. That means these three elements get exactly the same treatment in every browser:

**<input** type="text"**>**

**<input** type="super-strange-wonky-input-type"**>**

**<input>**

HTML5 uses this default to its benefit. It adds a few new data types to the <input> element, secure in the knowledge that browsers will treat them as ordinary text boxes if they don’t recognize them. For example, if you need to create a text box that holds an email address, you can use the new input type*email*:

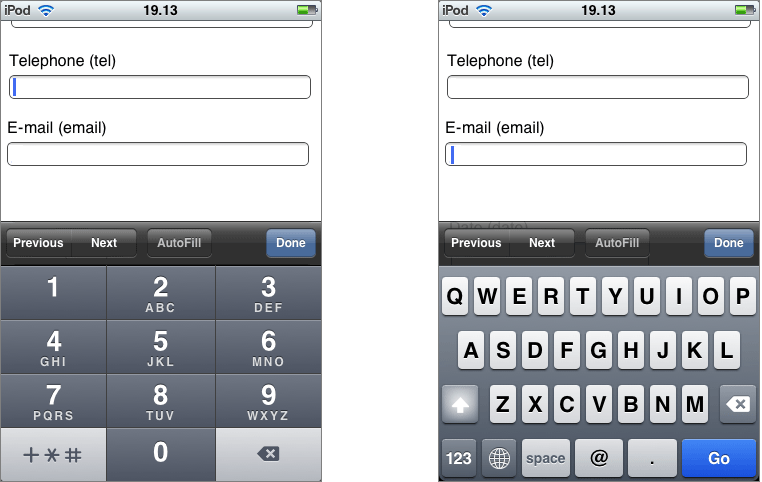
**<label** for="email"**>**Email **<em>**\***</em></label>**

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

If you view this page in a browser that doesn’t directly support the email input type (like Internet Explorer 9), you’ll get an ordinary text box, which is perfectly acceptable. But browsers that support HTML5 forms are a bit smarter. Here’s what they can do:

* **Offer editing conveniences**. For example, an intelligent browser might give you a way to get an email from your address book and pop it into an email field.
* **Restrict potential errors**. For example, browsers can ignore letters when you type in a number text box. Or, they can reject invalid dates (or just force you to pick one from a mini calendar, which is easier *and* safer).
* **Perform validation**. Browsers can perform more sophisticated checks when you click a submit button. For example, an intelligent browser will spot an obviously incorrect email address in an email box and refuse to continue.

The HTML5 specification doesn’t give browser makers any guidance on the first point. Browsers are free to manage the display and editing of different data types in any way that makes sense, and different browser can add different little luxuries. For example, mobile browsers take advantage of this information to customize the virtual keyboard that they show, hiding keys that don’t apply (see [Figure 4-8](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#when_people_use_a_mobile_device_to_fill_)).



*Figure 4-8. When people use a mobile device to fill out a form, they don’t have the luxury of entering information on a full keyboard. The iPod makes life easier by customizing the virtual keyboard depending on the data type, so telephone numbers get a telephone-style numeric keypad (left), while email addresses get a dedicated @ button and a smaller space bar (right).*

The error-prevention and error-checking features are more important. At a bare minimum, a browser that supports HTML5 web forms must prevent a form from being submitted if it contains data that breaks the data type rules. So if the browser doesn’t prevent errors (according to the second point in this list), it must validate them when the user submits the data (that’s the third point).

Unfortunately, not all current browsers live up to this requirement. Some recognize the new data types and provide some sort of editing niceties but no validation. Many understand one data type but not another. Mobile browsers are particularly problematic—they provide some editing conveniences but currently have none of the validation.

[Table 4-3](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#browser_compatibility_for_new_input_type) lists the new data types and the browsers that support them completely—meaning that they prevent forms from being submitted when the data type rules are broken.

*Table 4-3. Browser compatibility for new input types*

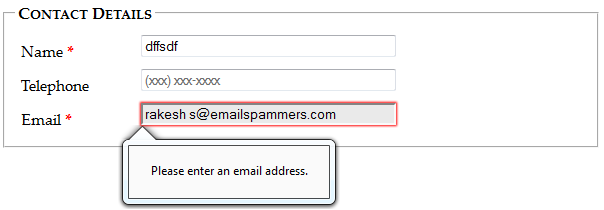
| **Data type** | **IE** | **Firefox** | **Chrome** | **Safari** | **Opera** | **Safari iOS[**[**a**](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#ftn.CHP-4-FNOTE-2)**]** | **Android** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| email | - | 4 | 10 | 5 (Windows only) | 10.6 | - | - |
| url | - | 4 | 10 | 5 (Windows only) | 10.6 | - | - |
| search[[b](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#ftn.CHP-4-FNOTE-3)] | n/a | n/a | n/a | n/a | n/a | n/a | n/a |
| tel[[c](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#ftn.CHP-4-FNOTE-4)] | n/a | n/a | n/a | n/a | n/a | n/a | n/a |
| number | - | - | 10 | 5 (Windows only) | - | - | - |
| range | - | - | 6 | 5 | 11 | - | - |
| datetime, date, month, week, time | - | - | 10 | - | 11 | - | - |
| color | - | - | - | - | 11 | - | - |
| [[a](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#CHP-4-FNOTE-2)]*Although the iOS browser doesn’t provide validation, its virtual keyboard customization (see* [Figure 4-8](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#when_people_use_a_mobile_device_to_fill_) *) is a significant convenience, so it’s still worth using specialized data types*.  [[b](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#CHP-4-FNOTE-3)]*The HTML5 standard does not require validation for this data type*.  [[c](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#CHP-4-FNOTE-4)]*The HTML5 standard does not require validation for this data type*. | | | | | | | |

TIP

Incidentally, you can test for data type support in Modernizr using the properties of the Modernizr.inputtypes object. For example, Modernizr.inputtypes.range returns true if the browser supports the range data type.

Email Addresses

Email addresses use the *email* type. Generally, a valid email address is a string of characters (with certain symbols not allowed). An email address must contain the @ symbol and a period, and there needs to be at least one character between them and two characters after the period. These are, more or less, the rules that govern email addresses. However, writing the right validation logic or regular expression for email addresses is a surprisingly subtle task that has tripped up many a well-intentioned developer. Which is why it’s great to find web browsers that support the email data type and perform validation automatically (see [Figure 4-9](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#firefox_refuses_to_accept_the_space_in_t)).



*Figure 4-9. Firefox refuses to accept the space in this spurious email address.*

Email attributes support the *multiple* attribute, which allows the field to hold multiple email addresses. However, these multiple email addresses still look like a single piece of text—you just separate each one with a comma.

NOTE

Remember, blank values bypass validation. If you want to force someone to enter a valid email address, you need the email data type combined with the required attribute ([Validation: Stopping Errors](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#validation_stopping_errors)).

URLs

URLs use the *url* type. What constitutes a valid URL is still a matter of hot debate. But most browsers use a relatively lax validation algorithm. It requires a URL prefix (which could be legitimate, like *http://*, or made up, like *bonk://*), and accepts spaces and most special characters other than the colon (:).

Some browsers also show a drop-down list of URL suggestions, which is typically taken from the browser’s history of recently visited pages.

Search Boxes

Search boxes use the *search* type. A search box is generally meant to contain keywords that are then used to perform some sort of search. It could be a search of the entire Web (as with Google, in [Figure 4-1](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#googles_spartan_search_page_holds_a_basi)), a search of a single page, or a custom-built search routine that examines your own catalog of information. Either way, a search box looks and behaves almost exactly like a normal text box.

On some browsers, like Safari, search boxes are styled slightly differently, with rounded corners. Also, as soon as you start typing in a search box in Safari or Chrome, a small X icon appears on the right side that you can click to clear the box. Other than these very minor differences, search boxes *are* text boxes. The value is in the semantics. In other words, you use the search data type to make the purpose of the box clear to browsers and assistive software. They can then guide visitors to the right spot or offer other smart features—maybe, someday.

Telephone Numbers

Telephone numbers use the *tel* type. Telephone numbers come in a variety of patterns. Some use only numbers, while others incorporate spaces, dashes, plus signs, and parentheses. Perhaps it’s because of these complications that the HTML5 standard doesn’t ask browsers to perform any telephone number validation at all. However, it’s hard to ignore the feeling that a tel field should at least reject letters (which it doesn’t).

Right now, the only value in using the tel type is to get a customized virtual keyboard on mobile browsers, which focuses on numbers and leaves out letters.

Numbers

HTML5 defines two numeric data types. The *number* type is the one to use for ordinary numbers.

The number data type has obvious potential. Ordinary text boxes accept anything: numbers, letters, spaces, punctuation, and the symbols usually reserved for cartoon character swearing. For this reason, one of the most common validation tasks is to check that a given value is numeric and falls in the right range. But use the number data type, and the browser automatically ignores all non-numeric keystrokes. Here’s an example:

**<label** for="age"**>**Age**<em>**\***</em></label>**

<input id="age" **type="number"**>**<br>**

Of course, there are plenty of numbers, and they aren’t all appropriate for every kind of data. The markup shown above allows ages like 43,000 and -6. To fix this, you need to use the min and max attributes. Here’s an example that limits ages to the reasonable range of 0 to 120:

**<input** id="age" type="number" **min="0" max="120"**>**<br>**

Ordinarily, the number data type accepts only whole numbers, so a fractional age like 30.5 isn’t accepted. (In fact, some browsers won’t even let you type the decimal point.) However, you can change this too by setting the *step* attribute, which indicates the acceptable intervals for the number. For example, a step minimum value of 0 and a step of 0.1 means you can use values like 0, 0.1, 0.2, and 0.3. Try to submit a form with 0.15, however, and you’ll get the familiar pop-up error message. The default step is 1.

**<label** for="weight"**>**Weight (in pounds)**</label>**

**<input** id="weight" type="number" **min="50" max="1000" step="0.1"**

**value="160"**>**<br>**

The step attribute also affects how the spin buttons work in the number box, as shown in [Figure 4-10](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#many_browsers_add_spin_buttons_to_number).



*Figure 4-10. Many browsers add spin buttons to number boxes. Every time you click the up arrow, the number increases by the step value (unless you reach the maximum allowed number). Similarly, when you click the down arrow, the number decreases by one step.*

Sliders

The *range* type is HTML5’s other numeric data type. Like the number type, it can represent whole numbers or fractional values. It also supports the same attributes for setting the range of allowed values (min and max). Here’s an example:

**<label** for="weight"**>**Weight (in pounds)**</label>**

**<input** id="weight" **type="range"** min="50" max="1000" value="160">**<br>**

The difference is the way the range type presents its information. Instead of asking you to type the value you want in a text box, intelligent browsers show a slider control ([Figure 4-11](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#youve_probably_used_range-like_controls_)).



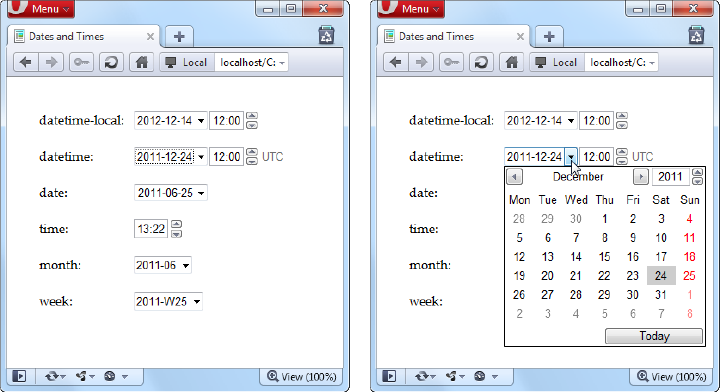
*Figure 4-11. You’ve probably used range-like controls to set things like sound volume. They work well for information where there’s a narrow range of values between a known minimum and a known maximum, and the specific numeric value isn’t important (but the placement of that value relative to the minimum and maximum is).*

To set a range type, you simply pull the tab to the position you want, somewhere between the minimum and maximum values at either end of the slider. Browsers that support the range type don’t give any feedback about the specific value that’s set. If you want that piece of information, you need to add a scrap of JavaScript that reacts when the slider changes (perhaps by handling the onChange event) and then displays the value nearby. Of course, you’d also want to check if the current browser supports the range type (using a tool like Modernizr). If the browser doesn’t support the range type, there’s no need to take any extra steps, because the value will show up in an ordinary text box.

Dates and Times

HTML5 defines several date-related types. Browsers that understand the date types can provide handy drop-down calendars for people to pick from. Not only does this clear away confusion about the right way to format the date, it also prevents people from accidentally (or deliberately) picking a date that doesn’t exist. And smart browsers can go even further—for example, adding integration with a personal calendar.

Right now, the date types are poorly supported, despite their obvious usefulness. Opera is the only browser that provides the drop-down calendars (see [Figure 4-12](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#the_ltinputgt_boxes_look_slightly_differ)). Chrome provides the bare minimum in support: It uses a text box with spin buttons, like the number type, and refuses bad dates.



*Figure 4-12. The <input> boxes look slightly different when storing date and time information (left). But the real convenience that Opera provides is the drop-down calendar that lets you set these values with a proper date, and no formatting headaches (right).*

TIP

If you decide to use one of the date types, you may want to use a polyfill for browsers that don’t understand it, like html5Widgets library described earlier ([www.useragentman.com/tests/html5Widgets](http://www.useragentman.com/tests/html5Widgets)). That’s because it’s easy for people on non-supporting browsers to enter dates in the wrong format, and it’s tedious for you to validate date data and provide the appropriate guidance. (That’s also why custom Java-Script date controls already exist—and why they’re all over the Web.)

[Table 4-4](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#date_data_types) explains the six date formats.

*Table 4-4. Date data types*

| **Date type** | **Description** | **Example** |
| --- | --- | --- |
| date | A date in the format *YYYY-MM-DD*. | January 25, 2012, is: *2012-01-25* |
| time | A 24-hour time with an optional seconds portion, in the format *HH:mm:ss.ss*. | 2:35 p.m. (and 50.2 seconds) is: *14:35* or*14:35:50.2* |
| datetimelocal | A date and a time, separated by a capital T (so the format is *YYYY-MM-DDTHH:mm:ss*). | January 25, 2012, 2:35 p.m:*2012-01-15T14:35* |
| datetime | A date and a time, like the datetime-local data type, but with a time-zone offset. This uses the same format (*YYYY-MM-DDTHH:mm:ss-HH:mm*) as the <time> element you considered on [Dates and Times with <time>](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch03.html#dates_and_times_with_lttimegt). | January 25, 2012, 2:35 p.m., in New York:*2012-01-15T14:35-05:00* |
| month | A year and month number, in the format*YYYY-MM*. | First month in 2012:*2012-01* |
| week | A year and week number, in the format*YYYY-Www*. Note that there can be 52 or 53 weeks, depending on the year. | Second week in 2012:*2012-W02* |

TIP

Browsers that support the date types also support the min and max attributes with them. That means you can set maximum and minimum dates, as long as you use the right date format. So, to restrict a date field to dates in the year 2012, you would write: *<input type=“date” min=“2012-01-01” max=“2012-12-31”>*.

Colors

Colors use the *color* type. The color data type is an interesting, albeit little-used, frill that lets a web page visitor pick a color from a drop-down color picker, which looks like what you might see in a desktop paint program. Currently, Opera is the only browser to add one. In other browsers, you need to type a hexadecimal color code on your own (or use the html5Widgets library, from [www.useragentman.com/tests/html5Widgets](http://www.useragentman.com/tests/html5Widgets)).

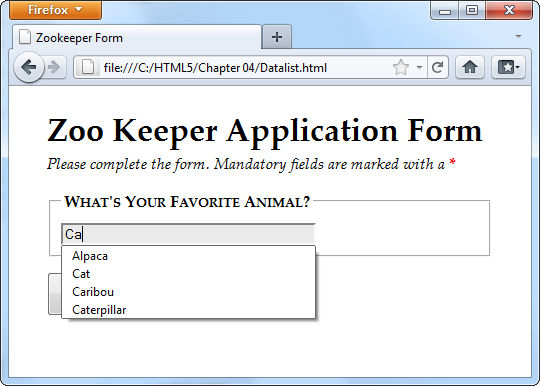
New Elements

So far, you’ve learned how HTML5 extends forms with new validation features, and how it gets smarter about data by adding more input types. These are the most practical and the most widely supported new features, but they aren’t the end of the HTML5 forms story.

HTML5 also introduces a few entirely new elements to fill gaps and add features. Using these nifty new elements, you can add a drop-down list of suggestions, a progress bar, a toolbar, and more. The problem with new elements is that old browsers are guaranteed not to support them, and even new browsers are slow to wade in when the specification is still changing. As a result, these details include some of the *least* supported features covered in this chapter. You may want to see how they work, but you’ll probably wait to use them, unless you’re comfortable inching your way out even further into the world of browser quirks and incompatibilities.

Input Suggestions with <datalist>

The <datalist> element gives you a way to fuse a drop-down list of suggestions to an ordinary text box. It gives people filling out a form the convenience to pick an option from the list, or the freedom to type exactly what they want (see [Figure 4-13](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#as_you_type_the_browser_shows_you_potent)).



*Figure 4-13. As you type, the browser shows you potential matches. For example, type in the letters “ca,” and the browser shows you every animal that has that letter sequence somewhere in its name (and not necessarily at the beginning).*

To use a datalist, you must first start with a standard text box. For example, imagine you have an ordinary <input> element like this:

**<legend>**What's Your Favorite Animal?**</legend>**

**<input** id="favoriteAnimal"**>**

To add a drop-down list of suggestions, you need to create a datalist. Technically, you can place that <datalist> element anywhere you want. That’s because the datalist can’t display itself—instead, it simply provides data that an input box will use. However, it makes logical sense to place the <datalist> element just after or just before the <input> element that uses it. Here’s an example:

**<datalist** id="animalChoices"**>**

**<option** label="Alpaca" value="alpaca"**>**

**<option** label="Zebra" value="zebra"**>**

**<option** label="Cat" value="cat"**>**

**<option** label="Caribou" value="caribou"**>**

**<option** label="Caterpillar" value="caterpillar"**>**

**<option** label="Anaconda" value="anaconda"**>**

**<option** label="Human" value="human"**>**

**<option** label="Elephant" value="elephant"**>**

**<option** label="Wildebeest" value="wildebeest"**>**

**<option** label="Pigeon" value="pigeon"**>**

**<option** label="Crab" value="crab"**>**

**</datalist>**

The datalist uses <option> elements, just like the traditional <select> element. Each <option> element represents a separate suggestion that may be offered to the form filler. The label shows the text that appears in the text box, while the value tracks the text that will be sent back to the web server, if the user chooses that option. On its own, a datalist is invisible. To hook it up to a text box so it can start providing suggestions, you need to set the *list* attribute to match the ID of the corresponding datalist:

<input id="favoriteAnimal" **list="animalChoices"**>

On browsers that supports the datalist—and right now, that’s only Opera 10 and Firefox 4 (or later)—visitors will see the result shown in [Figure 4-13](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#as_you_type_the_browser_shows_you_potent). Other browsers will ignore the list attribute and the datalist markup, rendering your suggestions useless.

However, there’s a clever fallback trick that makes other browsers behave properly. The trick is to put other content inside the datalist. This works because browsers that support the datalist pay attention to <option> elements only, and ignore all other content. Here’s a revised example that exploits this behavior. (The bold parts are the markup that datalist-supporting browsers will ignore.)

**<legend>**What's Your Favorite Animal?**</legend>**

**<datalist** id="animalChoices"**>**

**<span class="Label">Pick an option:</span>**

**<select id="favoriteAnimalPreset">**

**<option** label="Alpaca" value="alpaca"**>**

**<option** label="Zebra" value="zebra"**>**

**<option** label="Cat" value="cat"**>**

**<option** label="Caribou" value="caribou"**>**

**<option** label="Caterpillar" value="caterpillar"**>**

**<option** label="Anaconda" value="anaconda"**>**

**<option** label="Human" value="human"**>**

**<option** label="Elephant" value="elephant"**>**

**<option** label="Wildebeest" value="wildebeest"**>**

**<option** label="Pigeon" value="pigeon"**>**

**<option** label="Crab" value="crab"**>**

**</select>**

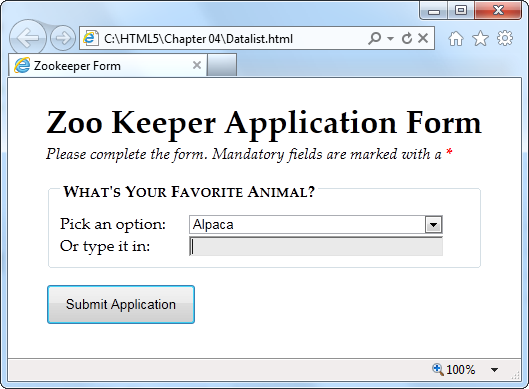
**<br>**

**<span class="Label">Or type it in:</span>**

**</datalist>**

**<input** list="animalChoices" name="list"**>**

If you remove the bold markup, you end up with the same markup you had before. That means browsers that recognize the datalist still show the single input box and the drop-down suggestion list, as shown in [Figure 4-13](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#as_you_type_the_browser_shows_you_potent). But on other browsers, the additional details wrap the datalist suggestion in a traditional select list, giving users the option of typing in what they want or picking it from a list ([Figure 4-14](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#you_can_still_use_your_suggestions_on_br)).



*Figure 4-14. You can still use your suggestions on browsers that don’t support the datalist. But you need to wrap them in a <select> list first.*

This effect isn’t completely seamless. When you receive the form data on the web server, you need to check for data from the list (in this example, that’s favoriteAnimalPreset) and from the text box (that’s favoriteAnimal). But other than this minor wrinkle, you’ve got a solid way to add new conveniences without leaving anyone behind.

NOTE

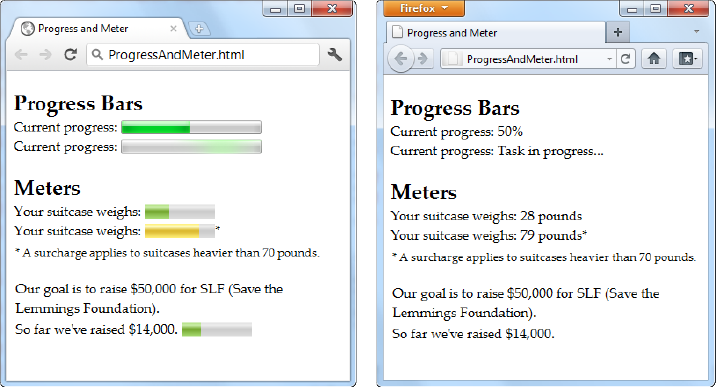
When the datalist was first created, it had a feature that it let it fetch suggestions from somewhere else—for example, it could call a web server, which could then pull a list of suggestions out of a database. This feature might still be added in a future version of the HTML standard, but for now it’s possible only if you write JavaScript code to handle the fetching, with the help of the XMLHttpRequest object ([The XMLHttpRequest Object](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch11.html#the_xmlhttprequest_object)).

Progress Bars and Meters

The <progress> and <meter> element are two new graphical widgets that look similar but serve different purposes ([Figure 4-15](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#the_ltprogressgt_and_ltmetergt_elements_)).

The <progress> element indicates how far a task has progressed. It uses a gray background that’s partially filled in with a pulsating green bar. The <progress> element resembles the progress bars you’ve probably seen before (for example, when the Windows operating system is copying files), although its exact appearance depends on the browser being used to view the page.

The <meter> element indicates a value within a known range. It looks similar to the <progress> element, but the green bar is a slightly different shade, and it doesn’t pulse. Depending on the browser, the meter bar may change color when a value is classified as “too low” or “too high”—for example, in the latter case Chrome changes the bar from green to yellow. But the most important difference between <progress> and <meter> is the semantic meaning that the markup conveys.



*Figure 4-15. The <progress> and <meter> elements are nice visual conveniences, on the browsers that support them (left). Other browsers simply substitute the fallback content you’ve set (right).*

NOTE

Technically, the new <meter> and <progress> elements don’t need to be in a form. In fact, they aren’t even real controls (because they don’t collect input from the web page visitor). However, the official HTML5 standard lumps them all together, because in some respects the <progress> and <meter> elements *feel* like form widgets, probably because they display bits of data in a graphical way.

Currently, the <progress> and <meter> elements work in Chrome (version 9 or later), Opera (version 11 or later), and Safari (version 5.1 or later). Firefox and Internet Explorer don’t support them yet.

Using the <progress> and <meter> elements is easy. First, consider the <progress> element. It takes a *value* attribute, which sets the percentage of progress (and thus the width of the green fill) as a fractional value from 0 to 1. For example, you could set the value to 0.25 to represent a task that’s 25 percent complete:

**<progress** value="0.25"**></progress>**

Alternatively, you can use the max attribute to set an upper maximum and change the scale of the progress bar. For example, if max is 200, your value needs to fall between 0 and 200. If you set it to 50, you’d get the same 25 percent fill as in the previous example:

**<progress** value="50" max="200"**></progress>**

The scale is simply a matter of convenience. The web page viewer doesn’t see the actual value in the progress bar.

NOTE

The <progress> element is simply a way to display a nicely shaded progress bar. It doesn’t actually *do* anything. For example, if you’re using the progress bar to show the progress of a background task (say, using web workers, as demonstrated on [Doing Work in the Background](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch12.html#doing_work_in_the_background)), it’s up to you to write the JavaScript code that grabs hold of the <progress> element and changes its value.

Browsers that don’t recognize the <progress> element simply ignore it. To deal with this problem, you can put some fallback content inside the <progress> element, like this:

**<progress** value="0.25"**>25%</progress>**

Just remember that the fallback content won’t appear in browsers that *do*support the <progress> element.

There’s one other progress bar option. You can show an *indeterminate*progress bar, which indicates that a task is under way, but you aren’t sure how close it is to completion. (Think of an indeterminate progress bar as a fancy “in progress” message.) An indeterminate progress bar looks like an empty gray bar but has a periodic green flash travel across it, from left to right. To create one, just leave out the value attribute, like this:

**<progress>**Task in progress ...**</progress>**

The <meter> element has a similar model, but it indicates any sort of measurement. The <meter> element is sometimes described as a *gauge*. Often, the specific meter value you use will correspond to something in the real world (for example, an amount of money, a number of days, an amount of weight, and so on). To control how the <meter> element displays this information, you’re able to set both a minimum and maximum value (using the *min* and*max* attributes):

Your suitcase weighs: **<meter** min="5" max="70" value="28"**>**28 pounds**</meter>**

Once again, the content inside the <meter> element is shown only if the browser doesn’t know how to display a meter bar. Of course, sometimes it’s important to show the specific number that the <meter> element uses. In this case, you’ll need to add it to the page yourself, and you don’t need the fallback content. The following example uses this approach. It provides all the information up-front and adds an optional <meter> element on browsers that support it:

**<p>**Our goal is to raise $50,000 for SLF (Save the Lemmings Foundation).**</p>**

**<p>**So far we've raised $14,000. **<meter** max="50000" value="14000"**></meter>**

The <meter> element also has the smarts to indicate that certain values are too high or too low, while still displaying them properly. To do this, you use the *low* and *high* attributes. For example, a value that’s above high (but still below max) is higher than it should be, but still allowed. Similarly, a value that’s below low (but still above min) is lower than it should be:

Your suitcase weighs:

**<meter** min="5" max="100" high="70" value="79"**>**79 pounds**</meter>**\*

**<p><small>**\* A surcharge applies to suitcases heavier than 70 pounds.

**</small></p>**

Browsers may or may not use this information. For example, Chrome shows a yellow bar for overly high values (like the one in the previous example). It doesn’t do anything to indicate low values. Finally, you can flag a certain value as being an optimal value using the *optimum* attribute, but it won’t change the way it shows up in today’s browsers.

All in all, <progress> and <scale> are minor conveniences that will be useful once their browser support improves just a bit.

Toolbars and Menus with <command> and <menu>

Count this as the greatest feature that’s not yet implemented. The idea is to have an element that represents actions the user can trigger (that’s <command>) and another one to hold a group of them (that’s <menu>). Depending on how you put it together and what styling tricks you use, the <menu> element could become anything from a toolbar docked to the side of the browser window to a pop-up menu that appears when you click somewhere on the page. But right now, no browser supports these elements, and so you’ll have to wait to find out if they’re really as cool as web developers hope.

An HTML Editor in a Web Page

As you learned in [Chapter 1](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch01.html), HTML5 believes in paving cowpaths—in other words, taking the unstandardized features that developers use today, and making them part of the official HTML5 standard. One of the single best examples is its inclusion of two odd attributes, named contentEditable and design-Mode, which let you turn an ordinary browser into a basic HTML editor.

These two attributes are nothing new. In fact, they were originally added to Internet Explorer 5 in the dark ages of the Internet. At the time, most developers dismissed them as more Windows-only extensions to the Web. But as the years wore on, more browsers began to copy IE’s practical but quirky approach to rich HTML editing. Today, every desktop browser supports these attributes, even though they have never been part of an official standard.

Using contentEditable to Edit an Element

The first tool you have for HTML editing is the *contentEditable* attribute. Add it to any element to make the content of that element editable:

**<div** id="editableElement" contentEditable**>**You can edit this text, if you'd

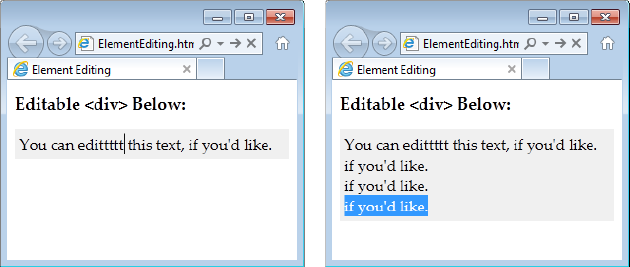
like.**</div>**

You probably won’t notice the difference at first. But if you load your page in a browser and click inside that <div>, a text-editing cursor (called a *caret*) will appear ([Figure 4-16](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#when_you_click_in_an_editable_region_you)).

**UP TO SPEED: WHEN TO USE HTML EDITING**

Before you try out rich HTML editing, it’s worth asking what the feature is actually for. Despite its immediate cool factor, HTML editing is a specialized feature that won’t appeal to everyone. It makes most sense if you need a quick-and-easy way for users to edit HTML content—for example, if you need to let them add blog posts, enter reviews, post classified ads, or compose messages to other users.

Even if you decide you need this sort of feature, the contentEditable and designMode attributes might not be your first choice. That’s because they don’t give you all the niceties of a real web design tool, like markup-changing commands, the ability to view and edit the HTML source, spell-checking, and so on. Using HTML’s rich editing feature, you *can* build a much fancier editor, with a bit of work. But if you really need rich editing functionality, you may be happier using someone else’s ready-made editor, which you can then plug into your own pages. To review some of the most popular options, read the blog post at <http://ajaxian.com/archives/richtexteditors-compared>.



*Figure 4-16. When you click in an editable region, you can move around using the arrow keys, delete text, and insert new content (left). You can also select text with the Shift key and then copy, cut, and paste it (right). It’s a bit like typing in a word processor, only you won’t be able to escape the confines of the <div> to get to the rest of the page.*

In this example, the editable <div> contains nothing but text. However, you could just as easily put other elements inside. In fact, this <div> element could wrap your entire page, making the whole thing editable. Similarly, you could use contentEditable on multiple elements to make several sections of a page editable.

TIP

Some browsers support a few built-in commands. For example, you can get bold, italic, and underline formatting in IE using the shortcut keys Ctrl+B, Ctrl+I, and Ctrl+U. Similarly, you can reverse your last action in Firefox by pressing Ctrl+Z, and you can use all of these shortcuts in Chrome. To learn more about these editing commands, and how you can create a custom toolbar that triggers them, see Opera’s two-part article series at <http://tinyurl.com/htmlEdit1> and <http://tinyurl.com/htmlEdit2>.)

Usually, you won’t set contentEditable in your markup. Instead, you’ll turn it on using a bit of JavaScript, and you’ll turn it off when you want to finish editing. Here are two functions that do exactly that:

**function** startEdit() {

*// Make the element editable.*

**var** element = document.getElementById("editableElement");

element.contentEditable = **true**;

}

**function** stopEdit() {

*// Return the element to normal.*

**var** element = document.getElementById("editableElement");

element.contentEditable = **false**;

*// Show the markup in a message box.*

alert("Your edited content: " + element.innerHTML);

}

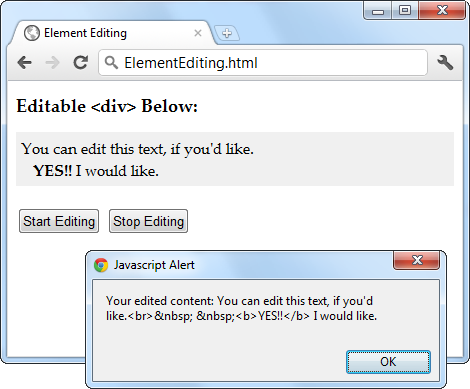
And here are two buttons that use them:

**<button** onclick="startEdit()"**>**Start Editing**</button>**

**<button** onclick="stopEdit()"**>**Stop Editing**</button>**

Just make sure you don’t place the buttons in the editable region of your page, because when a page is being edited, its elements stop firing events and won’t trigger your code.

[Figure 4-17](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#this_proves_itmdashediting_an_element_re) shows the result after the element has been edited and some formatting has been applied (courtesy of the Ctrl+B shortcut command).



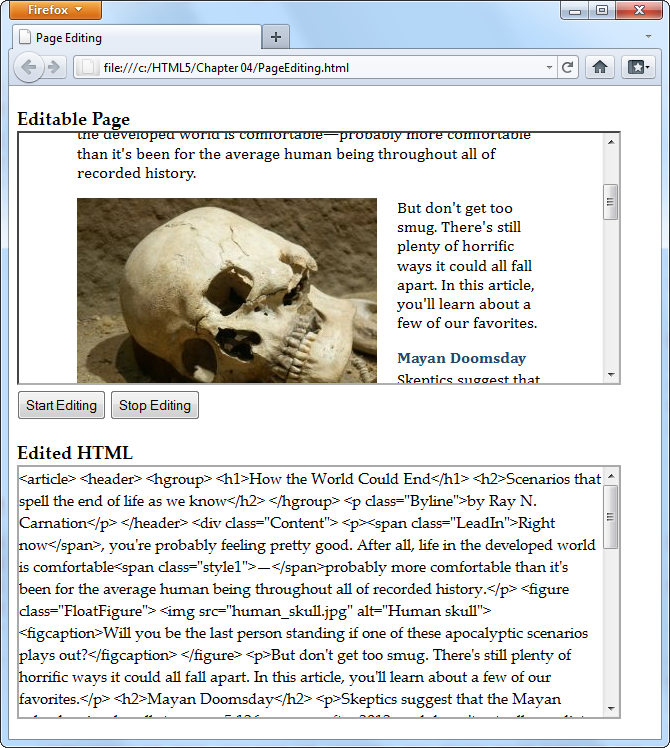
*Figure 4-17. This proves it—editing an element really does change the in-memory content of the page. In this example, the new content is simply shown in a message box. However, a more typical page would send this data to a web server, probably using the XMLHttpRequest object described on page 325.*

NOTE

There are subtle differences in the way rich HTML editing works in different browsers. For example, pressing Ctrl+B in Chrome adds a <b> element, while pressing it in IE adds the <strong> element. Similar variations occur when you hit the Enter key to add a new line or Backspace to delete a tag. One of the reasons that it makes sense for HTML5 to standardize the rich HTML editing feature is the ability to enforce better consistency.

Using designMode to Edit a Page

The designMode property is similar to contentEditable, except it allows you to edit an entire web page. This may seem like a bit of a problem—after all, if the whole page is editable, how will the user click buttons and control the editing process? The solution is to put the editable document inside an <iframe> element, which then acts as a super-powered editing box ([Figure 4-18](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch04.html#this_page_contains_two_boxes_the_first_i)).



*Figure 4-18. This page contains two boxes. The first is an <iframe> that shows the apocalypse page example from Chapter 2. The second is an ordinary <div> that shows the HTML markup of the page, after it’s been edited. The two buttons at the top of the page control the show, switching the <iframe> into design mode when the user is ready to work.*

The markup in this page is refreshingly simple. Here are the complete contents of the <body> element in the page:

**<h1>**Editable Page**</h1>**

**<iframe** id="pageEditor" src="ApocalypsePage\_Revised.html"**></iframe>**

**<div>**

**<button** onclick="startEdit()"**>**Start Editing**</button>**

**<button** onclick="stopEdit()"**>**Stop Editing**</button>**

**</div>**

**<h1>**Edited HTML**</h1>**

**<div** id="editedHTML"**></div>**

As you can see, this example relies on the startEdit() and stopEdit() methods, much like the previous example. However, the code is tweaked so that it sets the designMode attribute rather than the contentEditable attribute:

**function** startEdit() {

*// Turn on design mode in the <iframe>.*

**var** editor = document.getElementById("pageEditor");

editor.contentWindow.document.designMode = "on";

}

**function** stopEdit() {

*// Turn off design mode in the <iframe>.*

**var** editor = document.getElementById("pageEditor");

editor.contentWindow.document.designMode = "off";

*// Display the edited HTML (just to prove it's there).*

**var** htmlDisplay = document.getElementById("editedHTML");

htmlDisplay.textContent = editor.contentWindow.document.body.innerHTML;

}

This example gives you a better idea of the scope of the rich editing feature. For example, click on a picture and you’ll see how the browser lets you manipulate it. You can resize it, drag it to a new place, or delete it completely with a single click of the Delete button. You’ll have similar power over form controls, if they’re in the page you’re editing.

Of course, there’s still a significant gap you’ll need to cross if you want to turn this example into something practical. First, you’ll probably want to add better editing controls. Once again, the helpful folks at Opera have your back if you’re ready to make a deeper exploration into the command model, which is beyond the scope of this chapter (see <http://tinyurl.com/htmlEdit1> and <http://tinyurl.com/htmlEdit2>). Second, you’ll need to do something useful with your edited markup, like sending it to your web server via XMLHttpRequest ([The XMLHttpRequest Object](https://www.oreilly.com/library/view/html5-the-missing/9781449312671/ch11.html#the_xmlhttprequest_object)).

There’s one more caveat to note. If you run this example locally from your hard drive, it won’t work in all browsers. (Internet Explorer and Chrome run into security restrictions, while Firefox sails ahead without a problem.) To avoid the problem, you can run it from the try-out site at [www.prosetech.com/html5](http://www.prosetech.com/html5).

With Safari, you learn the way you learn best. Get unlimited access to videos, live online training, learning paths, books, interactive tutorials, and more.

[**START FREE TRIAL**](https://learning.oreilly.com/register/)