

CIS111-24171 (ONL) INFO SYSTEMS- PROGRAM/DB MGMT



Andrea Barrios
Coastline College

CIS 111-24171 (ONL) Info Systems- Program/DB Mgmt

Andrea Barrios
Coastline College

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (<https://LibreTexts.org>) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by [NICE CXone Expert](#) and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

This text was compiled on 05/15/2026

TABLE OF CONTENTS

Licensing

1: Document Markup

- 1.1: Markup Languages
- 1.2: Creating HTML Files
- 1.3: Page Layout
- 1.4: Text Layout
- 1.5: Navigation
- 1.6: Graphics
- 1.7: Tables
- 1.8: Forms
- 1.9: Canvas
- 1.10: Media Support
- 1.11: Mobile Device Support
- 1.12: Tags to Avoid
- 1.13: Rule Structure
- 1.14: Layout Formatting
- 1.15: Font and Text Decoration
- 1.16: Responsive Styling

2: CSS and Styling a Web Page

- 2.1: Web Page Header and Footer
- 2.2: Adding the form to the Page
- 2.3: Exercises

3: Trying It Together

- 3.1: Security
- 3.2: Integration Examples
- 3.3: Finishing Touches
- 3.4: Now What?

Index

Glossary

Detailed Licensing

Licensing

A detailed breakdown of this resource's licensing can be found in [Back Matter/Detailed Licensing](#).

CHAPTER OVERVIEW

1: Document Markup

Learning Objectives

By the end of this section, you should be able to demonstrate:

- the ability to create an HTML document structured to support CSS styling
- the ability to create a CSS file that adapts styling based on device capabilities
- the ability to create basic images using canvas
- the ability to integrate audio and video to a page
- the ability to utilize special device features
- the ability to integrate external font styles

[1.1: Markup Languages](#)

[1.2: Creating HTML Files](#)

[1.3: Page Layout](#)

[1.4: Text Layout](#)

[1.5: Navigation](#)

[1.6: Graphics](#)

[1.7: Tables](#)

[1.8: Forms](#)

[1.9: Canvas](#)

[1.10: Media Support](#)

[1.11: Mobile Device Support](#)

[1.12: Tags to Avoid](#)

[1.13: Rule Structure](#)

[1.14: Layout Formatting](#)

[1.15: Font and Text Decoration](#)

[1.16: Responsive Styling](#)

This page titled [1: Document Markup](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

1.1: Markup Languages

Document markup is a notation method that defines how particular pieces of information are meant to be formatted. The term comes from the practice of marking up manuscripts to notate changes that need to be made. Markup in terms of programming languages is used to identify a language that specifies how a document is to appear.

If you have ever used multiple colors of ink or highlighter when making notes and ascribed meaning to those colors for yourself (e.g., yellow highlighter is important, red ink is a definition) then you have already practiced document markup. You are providing additional layers of information along with the written text, in this case visual cues as to the purpose of the written information.

Some popular markup languages are hypertext markup language (HTML), extensible markup language (XML) and extensible hypertext markup language (XHTML). These were each created to fulfill particular needs in defining the layout and structure of the material.

HTML5



Hypertext markup language is used to aid in the publication of web pages by providing a structure that defines elements like tables, forms, lists and headings, and identifies where different portions of our content begin and end. It can be used to embed other file formats like videos, audio files, documents like PDFs and spreadsheets, among others. HTML is the most relied upon language in the creation of web sites. In this text we will focus on HTML5. While it is technically still in draft form, many proposed elements are already supported by the newer versions of most of the popular browsers.

History

In the beginning, back to the first days of the Internet and ARPA, the primary purpose of creating a page was to share research and information. HTML tags were only meant to provide layout and formatting of a page. As such, early implementations of HTML were somewhat limited as there was little demand for features beyond the basics. Headings, bullets, tables, and color were about all developers had to utilize. As sites were created for other more commercial uses, developers found creative ways of using these tools to get their pages looking more like magazines, advertisements, and what they had drawn on paper. Having been one of those developers, I recall the days of just-get-it-looking-right techniques, splicing page-sized images into tables so graphics were (usually) where we wanted them, nesting tables within tables to create complex layouts, and other methods that violate today's best practices.

Current State

While not formally finalized, many browsers are already supporting a number of features proposed in drafts of HTML5, including things like canvas and media support that greatly improve the browser's ability to process and display complex materials without requiring extensive coding and extensions. In the past, sites that used video and audio players had to integrate support for many players, and would have to include the libraries and formatted files for those systems in their sites. By providing a solution to using these media forms within HTML5, we can improve on the user experience and reduce the efforts necessary to provide them.

While these new features do reduce the amount of programming required to implement higher level elements, and include interactive elements that exceed document markup activities, HTML5 is still considered a markup language.

In these languages, we use tags to ascribe additional meaning to our text, which provide instruction to the browser as to how to display the text on the screen, but are not necessarily displayed to the user. In HTML and XHTML these tags are fixed, or predefined, meaning the names that can be used in the tags are limited to what browsers are able to recognize. In XML, tags are defined by the person creating the content as they are typically used in conjunction with data sources and provide information.

W3C Standards

The [World Wide Web Consortium](#),¹ or W3C, is an international community that supports web development through the creation of open standards that provide the best user experience possible for the widest audience of users. This group of professionals and experts come together to determine how CSS and HTML should operate, what tags should be included as features, and more. The W3C is also your best reference point in determining the accessibility of your site through the use of tools that analyze your code for W3C compliance. These tools confirm if you have fully implemented elements in your code, like providing alternate text descriptions of images in the event that the image cannot load, or the user is visually impaired.

In addition to the creation of accessibility standards, among many others, the W3C also provides tutorials and examples and is likely the most exhaustive reference you will find.

CSS



CSS stands for cascading style sheet, and is used to create rules about the color, font, and layout of our pages. It also determines when those rules are to be used, based on information like the device connecting to the page, or in response to a user's action. CSS can be used by not only HTML but any XML based language. By separating as much of the look and feel of a page from HTML as possible, we actually separate content from appearance. This makes it possible to quickly create several different versions of the appearance of our site, without recreating the content in each version. Our best approach is to use HTML to define the structure (and only structure) of our pages whenever possible, laying the groundwork for CSS to know where to apply the actual style.

History

As HTML grew in popularity, demands on its feature set also grew. Combined with the variety of browser implementations and their varied approaches to rendering and support, creating robust, visually appealing sites involved a significant amount of time and effort. To reduce these, and separate the duties of presentation from those of content, proposals were sought to define a new system of managing these features. CSS was born out of CHSS, or Cascading Hypertext Style Script, and extends our capabilities by allowing us to go far beyond the styling limits of HTML by giving us more power over images, making pages appear more newspaper or magazine-like with layout and typography affects, and reducing load time.

Introduced for public use in 1996, CSS1 contained the ability to apply rules by identifying elements (selectors), and most of the properties still in use today. CSS2 added the ability to adapt for different displays and devices, as well as positioning elements by specific values on the page. CSS2.1 followed with the introduction of additional features, but these were not considered substantial enough to warrant a full version number change.

Current State

While commonly referred to as CSS3, the numbering no longer applies to the language as a whole. The developers have decided to break the language into modules, allowing different aspects of the language to be revised and released independent of one another. This allows for stable modules to stay numbered as they are (since they are not actually changing), while those under more active development can be pushed out as needed. At the moment, most of the "current" modules are at version number 3. Some have not really changed from 2.1, while work on version 4 of selected modules is already underway.

Document Object Model

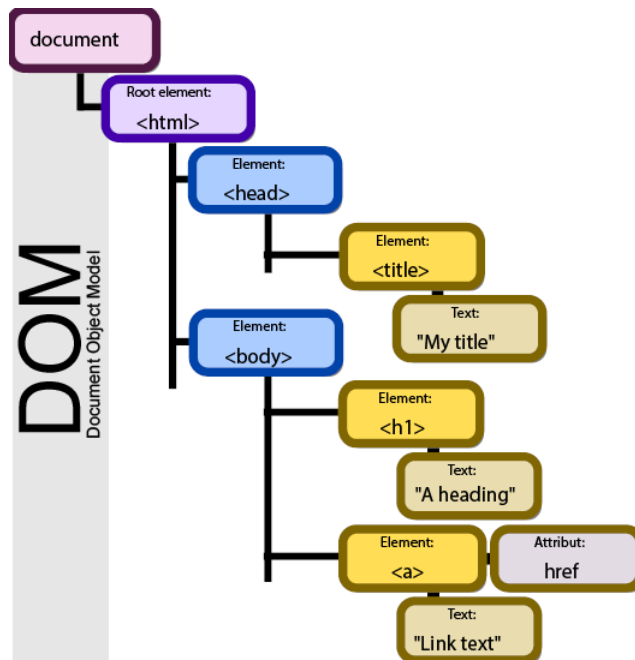


Figure 1.1.1: Document Object Model. (By Birger Eriksson CC-BY-SA-3.0)

Our ability to manipulate and create webpages consistently across formats comes from the document object model API, typically referred to as DOM. This API defines the order and structure of document files as well as how the file is manipulated to create, edit, or remove contents.

The DOM is built to be language and platform independent so any software or programming language can use it to interface with documents. It defines the interface methods and object types that represent elements of documents, the semantics and behavior of attributes of those objects, and also defines how they relate to one another. The DOM, effectively, is what gives rise to the tags we are about to study below. Languages that use the DOM, however, are not required to include all of its features, and may generate additional features of their own.

Figure 20 depicts an example of a document's model in tree format, with nested elements appearing to the right and below their parents. In this example, we are shown an HTML page with a section for the head and the body, which includes a page title and a link as its contents. This structure provides the ability for us to traverse, or move around the document, by referring to an object's name or attribute.

This page titled [1.1: Markup Languages](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.1: Markup Languages](#) by [Michael Mendez](#) has no license indicated.

1.2: Creating HTML Files

File Format

Before we can create our first web page, we need to create a file that our service will recognize as a web page. To do this, we can open our chosen text editor (see a short list of potentials in the section on [Integrated Development Environments](#)), and create a new document if one was not created automatically. We will immediately select “Save As” from your editor’s File menu, and give your new page its name. If this is going to be the front page, or first page you want a user to see for your site, you should name it index. Index is the default file name most web servers look for in any folder of your website; it saves your users from having to know and type the page name as part of the URL.

In many text editors, underneath or near where you enter the file name is another drop down selector that allows you to pick a file type. This is the extension (what comes after the period in the file name), or file type, that identifies what kind of data the file represents. This tells our operating system, applications, and browsers what conventions were used to create the content so it can be reassembled into usable form. Since we are creating a basic web page, we will use the .htm extension (.html is also acceptable, just be consistent to make your life easier). If your editor does not have .htm or .html in its list, then select “All” and make index.htm your file name.

Additional Notes

If you ever come across an unfamiliar extension and want to know more about it, sites like filext.com can help you determine what programs can open it and what it is for.

Once we have saved our file as index.htm, we are ready to begin. Saving as soon as we create a file is useful as the text editor will then know what syntax is expected. This will enable features like color coding and highlighting that your editor supports.

Document Type

Every HTML page we create should declare its document type (doctype) in the first line. This will identify which spec of HTML is included so the browser knows how to interpret the tags within. Earlier version of the HTML specifications used two definitions for HTML: HTML 4.01 and XHTML. Both of these contained two additional properties of strict and transitional.

With HTML5, much of this has been eliminated, leaving one general doctype declaration of `<!DOCTYPE html>`. This should be the first line of code in any HTML page you create. We will not cover the older doctype formats as all of our examples will focus on HTML5. Keep in mind, though, that code examples you find online with anything other than the tag above may be outdated approaches to what is shown.

Learn more

Keywords, search terms: xml, html, css, dom, document markup,

W3C Documentation for XML: <http://www.w3.org/TR/2004/REC-xml11-20040204/>

W3C Documentation for CSS: <http://www.w3.org/Style/CSS/>

W3C Specifications for HTML: <http://www.w3.org/community/webed/wiki/HTML/Specifications>

This page titled [1.2: Creating HTML Files](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.2: Creating HTML Files](#) by [Michael Mendez](#) has no license indicated.

1.3: Page Layout

Tags, Attributes

In order to add content to our page, we will set up our file with some basic structure. First, we will use tags to identify information about where parts of our page start and end. We will do this by using our first set of tags, `<html>` and `</html>`. We refer to these as a set because both use the same predefined word (HTML), and the latter uses the backslash (/) to indicate that it marks the close of that tag set. The act of placing a set of tags around content and other code is often referred to as “wrapping” what is between the tags. This is a good way to create a mental picture of what you are doing.

Next, we will put in a few more tags and then save our file and see some results. In between (typically referred to this as “inside”) your HTML tags, add an empty set of tags labeled head, and another labeled body. Then create two more sets labeled header and footer inside your body tags. Your resulting file should look like this:

1. `<html>`
2. `<head>`
3. `</head>`
4. `<body>`
5. `<header>`
6. `</header>`
7. `<footer>`
8. `</footer>`
9. `</body>`
10. `</html>`

Now we will add some spacing (see [Spacing](#) from our [Good Practices](#) section), and the ubiquitous Hello World in between the close of our header and start of our footer tags:

1. `<html>`
2. `<head>`
3. `</head>`
4. `<body>`
5. `<header>`
6. `</header>`
7. Hello World
8. `<footer>`
9. `</footer>`
10. `</body>`
11. `</html>`

Now we can better see how elements can be placed within one another, called nesting. If we save our file again, and double click the icon for our file in our documents folder to open it, it should open in your computer’s default browser as a white page with Hello World in the upper left corner. Congratulations! You have just made your first web page. Before you ask, though, unless you have created your own server, you will not be able to show your page to someone unless they are at your computer or you send them the file(s) you created. Since only the browser is needed to view the output of HTML, JavaScript, and CSS files, we will be able to view our pages without a server until the PHP examples later on.

As we continue to make changes, you can keep your editor and browser open at the same time. After you save your file in your text editor, you can simply refresh the page in your browser to see your changes.

Additional Notes

As we look at HTML elements, keep in mind that browsers will do their best to adjust for mistakes like missing tags. Different browsers will react to these problems in their own way, so just because it shows up correct in one browser does not mean it will in others.

Some tags, like those for including images, do not need a closing tag in order to work as no content is necessary inside the tags. In this case, we simply use the tag itself where we want it. In older versions of HTML one of the differences between document types was how we closed single tag elements. HTML, for example, wanted a break written as `<br>` while an XHTML document wanted `<br/>`. HTML5 will recognize either of these.

Head

Getting back into our document, we see that the first set of tags is called head. This is where we will put information about the page itself (called metadata) and where we will connect to any other resources like scripts and files that are not part of the page we are using. It is important that your `<head>` tags are always the first set of tags after `<html>`, as this is the order in which browsers expect the information.

To provide some basic information about our page or site, we will add some meta tags inside our head to give it a title, keywords, a description, and other details. These pieces of information help the user and bots understand what the page is about.

Each of these items are all parts of the meta tag (`<meta/>`). Some tags have special features that can be added to their definitions; these are called attributes. Meta tags support attributes like name and content. How we define these attributes will help the browser understand our page better. We will set our title as Our First Page, add the keywords beginning html, learning, and CSIT-107, and for a description we will put in a couple lines about what we are doing, and finally, add our name:

1. `<head>`
2. `<title>Our First Page</title>`
3. `<meta name="keywords" content="beginning html, learning, CSIT-107" />`
4. `<meta name="description" content="A beginning web page for CSIT-107 SUNY Fredonia" />`
5. `<meta name="author" content="Your Name Here" />`
6. `</head>`

There are a few things in this example for us to look at. First, you will note that title has its own tag. What you put in this tag is what will appear as the title in your browser window, and on the tab in your browser for this page (You can only have one set of these tags in a document). Keep in mind, this title will not appear on the page itself. If we want to include it as part of our page content, we would do that from within the body tags.

Next, we see that the values assigned to name and content are placed in quotes. The quotes identify what belongs to that attribute. While it is standard practice to use double-quoted strings, we can also use single quotes. These can be useful when creating HTML statements like those above as output from other languages (we will see examples of this in [PHP](#)).

You might also note that our keywords are comma separated, meaning we break them up by placing commas between different values; we do not need to add separate quotes around each word or phrase.

In the HTML5 specs, the meta tag can also have attributes named charset and http-equiv. Charset allows us to specify a particular set of characters we want to use for the page, an http-equiv supports content-type, default-style, and refresh as options. This lets us tell the browser what type of page we have (in our case, text/html), name a default style sheet (we will get to this later) and specify in terms of seconds how often we want to refresh the page, if at all.

Since our content will not be changing unless we change the page ourselves, and we do not have a style sheet yet, we will just add our content type declaration for now:

1. `<meta http-equiv="content-type" content="text/html" />`

Meta is also where we will define cache items like how long our page can be cached before it expires, or if we even want content to expire, as in these examples:

1. `<META HTTP-EQUIV="EXPIRES" CONTENT="Mon, 22 Jul 2002 11:12:01 GMT">`
2. `<META HTTP-EQUIV="CACHE-CONTROL" CONTENT="NO-CACHE">`

Body

All of the content that we wish to have on the screen should be encompassed by a set of body tags. The header, content, footer, and div (an all-purpose tag derived from “division”) sections are examples of what we can put in our body tags, which we will see in

the next example. Keep in mind that using these tags does not prevent us from seeing content that is not within the body. Tags are used to guide the interpreter in how to display the document. Unless it is specifically within the `<head>` tags, browsers may elect to display any content that is not properly nested in different ways.

Header, Footer

The header and footer tags are new in HTML5. They were added due to the volume of sites that define a top and bottom section to their pages. Allowing these tags makes it easier to define and find those parts of the layout. The header and footer should be nested within your body tags, but are not a requirement. For our example, we will put a screen title in our header and a copyright in our footer:

1. `<body>`
2. `<header>`
3. `<h1>This is our first page!</h1>`
4. `</header>`
5. Hello World
6. `<footer>`
7. `© 2013 Your Name Here`
8. `</footer>`
9. `</body>`

After saving your file and refreshing your browser, you should see our sentence in the top left, followed by Hello World, and then © 2013 Your Name Here. We made our title text extra-large by wrapping it in `<h1>` tags. H1 stands for heading one, the largest heading by default. We can also use h2 through h6 to access additional styles. Just as in a written document, we use these headings to distinguish different portions of our text. Your browser is applying a default style to make h1 look as it does on your screen. Later, we will see how to override this default style to make our headings look how we want them to. Using these headings allows us to quickly identify different portions of content not only for the reader, but also for search engines, which typically consider content in these tags as indicators of what your site is about, reducing our SEO efforts later.

Div/Span

While `<p>` helps us split up our text, we also need a mechanism to separate different pieces of content like we did when we used header and footer. This will allow us to define more than just a top, middle and bottom to our page. To do this, we can wrap those sections in `<div>` tags. Div stands for divide—it defines a section of content that should be treated as separate from other content. Span is very similar to div, except that it should identify inline content, meaning material that is within a block of text. Ultimately, a div will place a line break before and after its tags, while a span will not. Aside from this, these tags are functionally equivalent. While these tags seem very plain now, they are very useful when creating complex layouts, and are the tags we will use most often.

While div and span are effective for styling, we should strive to use the best set of tags available so browsers, users, and bots are able to understand our site and its layout, like the `<header>` and `<footer>` that we have already used in our code. Just keep in mind, not all of these are supported in all browsers. If you are wondering which browsers will work with what tags, you can refer to caniuse.com to see what is available.

To create organization for our site, we will add some more conceptual sections to our file. You will notice the layout does not actually go left or right as we are labeling our divs (everything will organize top-to-bottom). This is because we need to add CSS for the full effect. We will pick this example up again later to add the CSS needed to create the layout we want:

1. `<body>`
2. `<header>`
3. `<h1>This is our first page! </h1>`
4. `</header>`
5. `<div id="left">`
6. some menu items
7. `</div>`
8. `<div id="content">`

9. Hello World
10. </div>
11. <div id="right">
12. and some content on the right
13. </div>
14. <footer>
15. © 2013 Your Name Here
16. </footer>
17. </body>

This page titled [1.3: Page Layout](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.3: Page Layout](#) by [Michael Mendez](#) has no license indicated.

1.4: Text Layout

Text Layout

While all of the following tags are supported in HTML5, some of them are exclusive to the new specification, and will require a browser that is HTML5 compatible to work correctly.

Paragraphs

To build upon our basic structure a bit, we can break a long section of text into paragraphs. We can do this by adding breaks (`<br/>`) in our code. If we are going to do this a number of times, and if we want to style our paragraphs down the line, we should instead wrap each one in a set of paragraph tags, `<p></p>`. Using the paragraph tags allows us to automatically add spacing around our content to separate it from the rest of the page.

Before:

- [illegible]

After:

1. `<body>`
2. `<p>`
3. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document.
4. `</p>`
5. `<p>`
6. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document.
7. `</p>`
8. `<p>`
9. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document.

10. `</p>`
11. `</body>`

Ordered, Unordered Lists

If you are not familiar with the protocol for lists, unordered lists are intended for items that relate, but do not need to be in a particular order. Ordered lists, on the other hand, are for items that need to be ordered for a reason, like instructions that need to be followed in correct sequence.

Ordered and unordered lists are alphanumeric and unordered lists of items, respectively. Using these we can create lists to have them display on the screen as we are used to seeing a list of items, or we can take a list we have defined and use it as a group of common objects or ideas to build things like menus and navigation when we add additional CSS.

When we use the tags `<ol>` (ordered lists) or `<ul>` (unordered lists), we placed nested `<li>` tags in each to represent each item in the list.

• <code></code> • <code>First</code> • <code>Second</code> • <code>Third</code> • <code></code>	1. First 2. Second 3. Third
• <code></code> • <code>An item</code> • <code>Another item</code> • <code>Yet another item</code> • <code></code>	• An item • Another item • Yet another item

Definition Lists

A related set of tags can be used when you want to list definitions. These are `<dl>` for the list itself, with `<dt>` nested inside for terms and `<dd>` also nested, for definition, following its corresponding `<dt>`.

• <code><dl></code> • <code><dt>Coffee</dt></code> • <code><dd>Bean-based caffeinated beverage </dd></code> • <code><dt>Tea</dt></code> • <code><dd>Leaf-based caffeinated beverage</dd></code> • <code><dt>Water</dt></code> • <code><dd>Standard H2O</dd></code> • <code></dl></code>	Coffee Bean-based caffeinated beverage Tea Leaf-based caffeinated beverage Water Standard H2O
---	---

Address

The address tag allows us to specify text that belongs to an address or contact information for the content creator, making it easier for applications to find the information needed for tools like mapping and generating references.

1. `<address>`
2. Article by `<a href="mailto:professor@school.edu">Prof. Essor</a>.<br>`
3. Fredonia, NY`<br>`
4. USA
5. `</address>`

Article by [Prof. Essor](#).
Fredonia, NY
USA

Article

Article tags are meant to be used on content that can be re-used outside of its original site. It is meant for news articles, blog posts, and other types of content that would be republished in multiple locations.

1. `<article>`
2. `<h1>Our Blog Post</h1>`
3. `<p>This is our great content that is now identified as something that can exist on its own as a piece of work.</p>`
4. `</article>`

Aside

The aside is intended for use when you want to mark a piece of content that is related to the material it is nested within. It created primarily to define related information, like part of an article or blog.

1. `<p>`
2. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document. This is some text. It is really long. We want to break this into paragraphs so it looks more like a document.
3. `</p>`
4. `<aside>`
5. `<h4>Side Bar</h4>`
6. `<p>This is something related to our content that is not actually a part of it</p>`
7. `</aside>`

Button

A button is similar to the submit button, but unlike other input styles, it can include text or an image. Its default formatting gives it a beveled button appearance.

- `<button type="button">Click Here!</button>`
- `<button type="button">`
- `</button>`

 *Image of a button with a click here tag*

Caption

The caption tag is for tables, and allows you to define a label to be printed near the table for reference. You can only have one caption per table, and it must be after the opening table tag.

- `<table>`
- `<caption>This is our table</caption>`
- `<tr>`
- `<td>First Col</td><td>Second Col</td><td>Third Col</td>`
- `</tr>`
- `<tr>`
- `<td>1</td><td>2</td><td>3</td>`
- `</tr>`
- `<tr>`
- `<td>4</td><td>5</td><td>6</td>`
- `</tr>`
- `</table>`

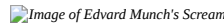
This is our table

First Col	Second Col	Third Col
1	2	3
4	5	6

Cite

While cite has been included in previous versions of HTML, the current HTML5 specification intends for it to be used to define the title of a work that is included in the document. Previous versions limited this tag to proper citations of written publications.

- ``
- `<p><cite>The Scream</cite> E. MunchChapter 1893.</p>`



Entities

In the examples in the [Header](#), [Footer](#) section, we placed a copyright symbol on the screen using “©” which told the browser what symbol we wanted to use. This is a reserved symbol, or entity, in HTML. We can call entities by using `&[entity name here];` or `&[entity number here];`. For example, ` ` means non-breaking space, or just a standard space. This is one way to insert extra spaces in our output. Since the browser would skip all the extra spaces in our code, we can add non-breaking space entities to tell the browser we do want it displayed on the screen.

The table below includes examples of other popular symbols (there are [plenty more](#)). Keep in mind, when you use entity names, they are case sensitive.

Table 1 HTML Entities

Result	Description	Name	Number
	non-breaking space	<code>&nbsp;</code>	<code>&#160;</code>
<code><</code>	less than	<code>&lt;</code>	<code>&#60;</code>
<code>></code>	greater than	<code>&gt;</code>	<code>&#62;</code>
<code>&</code>	ampersand	<code>&amp;</code>	<code>&#38;</code>
<code>©</code>	copyright	<code>&copy;</code>	<code>&#169;</code>
<code>®</code>	registered trademark	<code>&reg;</code>	<code>&#174;</code>
<code>™</code>	trademark	<code>&trade;</code>	<code>&#8482;</code>

Figure

The figure tag allows us to label an image, portrait, or other visual art included in an image tag to identify the content as such.

1. `<figure></figure>`

Figcaption

Figcaption, like caption, allows us to add a caption to our image like we would for a table.

1. `<figure>`
2. ``
3. `<figcaption>Figure 1</figcaption>`
4. `</figure>`

Mark

Most text altering tags have been skipped in this text as they can be easily achieved through CSS (and to maintain separation of duties). However, the mark tag is worth a look as an easy way to achieve a highlighting effect. It can be useful to insert this tag when generating output for things like search results.

- `<p>Here is a sentence with some <mark>highlighted</mark> text.</p>`

Here is a sentence with some highlighted text.

Meter

The meter tag allows us to generate a visual image based on provided values. This is intended for values that are already known or loaded to the screen like a chart or graph. There is also a [Progress](#) tag for monitoring file actions in progress like a download.

- `<meter value="3" min="0" max="15">One Fifth</meter><br>`
- `<meter value="0.65">65%</meter>`



Nav

If we have a group of links we want in one place (i.e. a menu or list of references), we can include them inside nav tags so browsers recognize them as a group of links. This is especially useful for screen reader software, as the tags provide an indicator as to what the links are for.

1. `<nav>`
2. `<a href="/">Home</a> |`
3. `<a href="/css/">CSS</a> |`
4. `<a href="/js/">JavaScript</a> |`
5. `<a href="/js/jquery/">jQuery</a>`
6. `</nav>`

Progress

The progress tag was created to help display the status of an upload or download. It takes two attributes including the current amount (which we would change using JavaScript) and the total or highest value of what we are monitoring. If we are showing the percentage of an upload we might use:

- `<progress value="46" max="100"></progress>`



Or, if we want to show the actual amount moved, or are moving a number of items, we can use the number completed and the total number instead of a percent, and the image will calculate it for us:

- `<progress value="345" max="850"></progress>`



Time

Another new-to-HTML5 element is time. The time tag is flexible in that it can specify a 24 hour formatted value, a full Gregorian calendar date, or both a date and time. The use of this tag by itself will not change any visual styling on the page, but allows applications on our devices to find the information in order to support features like creating calendar entries or reminders based on the information.

1. `<p>The daily meeting will be at <time>10:00</time> every morning.</p>`
2. `<p>The next monthly meeting will be on <time datetime="2013-08-01">August first</time>.</p>`

This page titled [1.4: Text Layout](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.4: Text Layout](#) by [Michael Mendez](#) has no license indicated.

1.5: Navigation

A feature found on almost every website is a navigation system for moving between pages. These are typically menus, where groups of common pages are created to give the site a hierarchical organization. While the approach to visual styling and interaction with menus comes in great variety, most follow a basic principle of using unordered lists of links, and the application of CSS to those lists in order to turn them into the colorful, interactive elements we are accustomed to. While there are drawbacks that we will discuss in [Visually Impaired Considerations](#), alternative approaches can still utilize linked lists to some extent.

Since we created our menu earlier, we already know the contents and structure of our navigation. Our group label, or top-level labels, and the nested `<ul>`s represent the contents of the list for that menu item.

Some popular approaches to providing a means of navigation are menu bars with drop downs, bread crumbs, and event driven responses. Menu bars are the most frequently utilized element, where hovering or clicking over an item in the menu brings up additional choices related to the main item. Typically referred to as drop down menus, they can be styled to move in any direction. Nesting lists within lists can give us a multi-tier menu that allows us the ability to select from a large number of pages with little effort.

Breadcrumbs are typically depicted as a horizontal delimited list of pages, similar to:

1. Home >> Sports >> Football >> Buffalo Bills >> Patriots >> Golf

The breadcrumb does not follow a hierarchical notation, but acts more like a brief history of where you have been on the site, allowing you to skip back several steps at once without using your browser's back button. These can be helpful in sites with large amounts of content where the user's experience may not be particularly linear, as they move between topics or sections, like news or reference sites.

Event-driven navigation is useful in narrowing the user experience to a fixed set of paths. This method will only make certain links available under certain conditions, restricting the options a user has on a particular page to what they are allowed to do, which may be based on a variety of rules such as if they are logged in, previous links or decisions they have made, or if something in the system needs their attention.

These approaches can be used by themselves, or in combination to provide your user experience.

Linking

Links in HTML can take two forms, both of which are created with the anchor tag (`<a>`). They can either point to a resource in another location, or to a location within the document. The former are used far more frequently than the latter, however internal links are coming back into popularity with the rise of infinite scrolling.

Absolute, Base, and Relative Path

The href attribute of an anchor tag defines the actual location the link will represent. Absolute and relative paths are two reference methods for connecting sites and pages. While both methods can be used when creating links that point to content in our own site, only absolute can be used when pointing to content that is outside of your domain.

Absolute paths are the entire length of the link required to identify one resource, whether it is a page, image, script, or media file. The URL <http://www.msn.com/news/index.htm> tells us we want to go to the index page in the news folder of the msn.com website. If this was our site, and we wanted to go to the index.htm file in the sports folder, we could write it as <http://www.msn.com/sports/index.htm> (absolute) or [../sports/index.htm](#) (relative). The initial `..` instructs the browser that our intention is to go back one layer of depth (i.e. "up" one level in folders) and then into the sports folder, which in this example sits in the same parent folder as our news page.

Using just an initial `/` without `..` tells the server that we want to start at the root folder of the server and navigate from there, meaning we start with the base path.

A base path is everything needed to get us to the index page of the root folder of the site. This is typically <http://www.yoursitename.com>, and is the part you find missing in the relative path above. The combination of the base path, and relative path, equals your absolute path.

Target

While the anchor tag supports several attributes, one of the most important of these is “target.” This attribute describes where links will be loaded, like a new tab or the same tab/browser window we are already using. The attribute can take any of the following values to define that location.

Table 1.5.1: Anchor Targets

Value	Description
_blank	Opens the linked document in a new window or tab
_self	Opens the linked document in the same frame as it was clicked (this is default)
_parent	Opens the linked document in the parent frame
_top	Opens the linked document in the full body of the window
framename	Opens the linked document in a named frame

From php.net manual, creative commons 3.0 Attribution

Within the Page

We can add links to a page that move the user around the page itself, which is useful on pages with long content. To do this, we use an anchor tag to define where we want our destination to be. When we create our link, we simply reference the name of our anchor, preceded by a pound sign in place of a traditional URL.

• Some text here. • <code>Click here to go further down.</code> • Some more text. • Even more text! • <code></code> • This is where we want to "jump" to.	Some text here. Click here to go further down. Some more text. Even more text! This is where we want to “jump” to.
--	---

This page titled [1.5: Navigation](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.5: Navigation](#) by [Michael Mendez](#) has no license indicated.

1.6: Graphics

Images are the greatest contributors to the visual appeal of your site, and typically account for the majority of bandwidth used in loading your pages. By using a combination of image types, and newer techniques found in HTML5 like canvas, and reproducing images using CSS, we can balance quality against size to reduce our bandwidth needs and allow our site to be more flexible.

Formats

Images are files, just like any other document in your computer, but they can be coded and formatted differently to reproduce the image you want to see. We find these referred to as raster and vector graphics. These formats represent two very different methods of creating an image.

Raster

The image files most of us are already familiar with using are typically raster format. Examples of these are JPEG, GIF and BMP. When we interact with pictures we took on digital cameras for example, we are dealing with JPEG or JPG files. Raster files recreate an image by recording the color value of pixels, which represent the smallest single point on a screen that can be assigned a color by the display. The higher the number of pixels (or density, measured as pixels per inch) translates to how sharp the image is, and how large it can be rendered without losing quality.

The number of colors available in the image file is based on the length of the value available to each point. If we only allowed a single binary character for each pixel point, we would be able to keep our file size as small as possible. This however would mean we could only represent our image in black and white (binary only allows us two options, 0 or 1, so we can only represent two colors.). When we allow longer values to represent a single point, we can assign values a larger range of colors. Once we scale these up, however, we trade away our smaller image sizes in order to have more colorful pictures. Large images can slow down the user experience, and if loading takes too long, users will leave.

Traditionally, we have faced this trade off by using different image formats in different areas of our site. While reserving JPG for our larger images or photos, we can use GIF for smaller icons and indicators. GIFs limit us to 256 colors, but since most icons use few colors, we are able to capitalize on the benefits of this format here. It is important to note that raster images will quickly lose quality when rendered at sizes larger than the original image's width or height.

Vector

Vector images store information about colors and locations as definitions of angles, lines, and curves in mathematical format. The benefit of a vector formatted image is that it can be scaled both up and down in size without distortion or degradation in quality. This is due to the fact that the image is “drawn” by the browser each time it is loaded, and the processor performs the steps necessary to recreate the image. Since the image can be scaled, the same image file can be drawn very large, or very small, without changing the file size. We will get some hands-on experience in how vector images are drawn when we look at the new [Canvas](#) features in HTML.

Table 1.6.1: Image Formats

Format	Compression	Platforms	Colors	Notes
JPEG (Joint Photographic Experts Group)	Lossy	Unix, Win, Mac	24-bit per pixel; 16.7 million colors.	JPEG is a compression algorithm; the format is actually JFIF (JPEG File Interchange Format)
GIF (Graphic Interchange Format)	Lossless	Unix, Win, Mac	8-bit; 256 colors (216 web palette). Allows transparency.	LZW compression algorithm developed by CompuServe; patent now held by Unisys, which charges for use of the code in graphics programs. Once Unisys began enforcing its patent (in 1995), programs began moving to PNG.
BMP (Bitmap graphics)	Uncompressed	Win	24-bit; 16.7 million colors.	Like all uncompressed formats, these files are very large.
PICT	Lossless	Mac		Very little compression; large files
TIFF (Tag Interchange File Format)	Lossless or uncompressed	Unix, Mac, Win		TIFF-LZW uses the proprietary LZW compressions (see GIF).
PNG (Portable Network Graphics)	Lossless	Unix, Mac, Win	48-bit; “true color” plus transparency	Will likely replace GIF. Supported in IE, NN 4 and above. A WC3specification .

<http://mason.gmu.edu/~dtaciuch>

You may notice the compression column. This is the act of removing or modifying the data that represents a file in a manner that makes its overall file size smaller. By doing this, we can transmit files faster, and they will take up less space in memory. When we discuss compression in terms of graphics we need to consider whether it will result in a lossy or lossless result. A lossless result means the compression techniques used do not remove data from the original copy, so we can restore the image to its exact original size and appearance. A lossy compressions structure can result in greater compression, but achieves the extra advantage by removing information from the file.

As an example, imagine a picture of you and your friends on the beach with a clear blue sky behind you. The data in the image file will measure the “blueness” of the sky in varying colors of blue, at a level greater than the eye can distinguish. By averaging the blueness and making more of the sky pixels the same color, we have eliminated information. Certain levels of lossy compression will still be indistinguishable from the original, but at any level, the lossy-compressed version of the file will not be restorable to the original because the extra values were eliminated.

Which is better? As usual, it depends on your intent. If the image can be lossy compressed and is still acceptable to you and your users, and having the smallest possible file sizes (good, of course, for mobile devices) is a priority, then go for it. Quality optimized scenarios will likely call for a lossless compression, like in sites that use large images as their background.

Slicing

For some time, there has been a practice of breaking larger images up into many smaller ones (a process called slicing), in an effort to allow pages to load more quickly. While this gave a visual experience of faster speed (each small image blinking into place as it was loaded) the load time was about the same, if not longer, as the overall file size had not changed, but we instead asked for it over multiple requests instead of waiting for the entire image in one request.

The need for this approach has been largely eliminated by modern versions of CSS (and other techniques we will discuss). This allows us to reproduce many things we used images for (i.e. buttons, hover effects, etc.) without using images at all, and allows us to have the control over layout and formatting that slicing an image used to fulfill. Now a common goal for site developers is to be as “imageless” as possible, using images only where CSS cannot stand in. This reduces load times and gives greater flexibility in site design. As an example of what can be done using CSS3, take a look at this simulated iPhone: <http://tjrus.com/iphone#71d465!>

Some additional techniques to reduce image weight on a site are right-sizing images, compression (which we just discussed above), caching, and sprites, among others. Right-sizing is editing and creating a copy of an image to the exact size needed where it is shown. For example, small images for items on a product page could simply be the original image rendered at a smaller size. If the user does not look at those products, loading the larger images first only degrades their experience, since every product’s large image file needs to be downloaded. Right-sizing and compression both require image editing software with at least some advanced editing features, or use of an online service that covers the basics (with less control on your part) like <http://www.imageoptimizer.net>.

The process of caching can also help. When your site is completed and your images (and other files) will persist, the use of caching can reduce load times for repeat visitors. Caching means the files the user downloads are marked in the server with an expiration. The next time the user visits the site, their device will check expiration times on the content. If the device’s local copy is not expired, it simply uses the one it has, without having to download it again.

The last option we will consider here (there are more, advanced methods) is creating a sprite. Almost the reverse of compression and right sizing, a sprite is one large image that contains all of the icons used throughout the site. Since these smaller images are often repeated, we will download one main image a single time. By using CSS rules to reveal only small piece of the image (i.e. the portion containing one icon) at a time, all of our icons can live in one image file but appear as individual elements on the page.

Favicon

A favicon is a special type of image. It is the small icon that accompanies the page title in the browser’s title and tab. This icon is automatically used on each page found in the folder the favicon is stored in. For example, to apply a single icon to your entire site you would place it in the root folder. Any folders below that level can use a different favicon.

These icons are usually 32 by 32 pixel images that represents the site or site’s parent company. They are converted to a special format and saved with the extension .ico to identify themselves as site icons. While they are small and provide little to the overall visual experience of a website, sites lacking a favicon tend to appear less legitimate as the icon space will be replaced by the browser’s default icon.

Creating favicons can be done in paint or photo editing software that allows you to comply with the size and color density limits of favicons. Additionally, sites like favicon-generator.org or www.favicon.cc among others can help convert existing images into favicons with some basic editing options before saving your new icons.

This page titled [1.6: Graphics](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.6: Graphics](#) by [Michael Mendez](#) has no license indicated.

1.7: Tables

Tables are a method of formatting the content of your page, and are very similar to the concept of a spreadsheet. Tables are composed of rows and columns. Each intersection of the two is referred to as a cell, and is where content is placed. The number of columns and rows you use depends on your need and design.

You will probably find a great number of sites that rely heavily on tables to create the look and feel of their pages. Recognizing I have already admitted to doing this in the past when necessary, I will repeat my earlier statement: Please, do not.

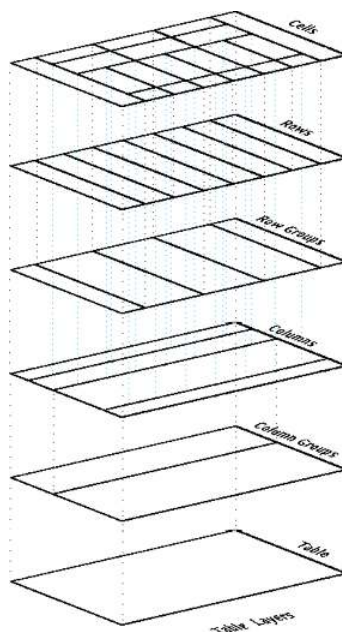


Figure 1.7.1: Table Structure. (By [Tarikash \(Own work\)](#)CC-BY-SA-3., via [Wikimedia Commons](#))

While this was a common method in the past, we now have elements in HTML that are defined for such purposes. Tables should be reserved for creating collections of data or information on your page and nothing else. If you are using them to store information other than how you would in a spreadsheet, you should be using a `div` or `span`. Using a table for content organization will create several complications in your code that can be reduced or eliminated by following today's conventions.

First, the formatting of the table is more closely tied to HTML as it is easier to define there than in CSS. Placing these definitions in our HTML breaks our goal of separation of duties to support a responsive design. If we wanted to change the layout, we would have to edit our HTML instead of just our CSS files. Second, it creates extra lines of code to define the layout we are looking for, and the use of column and `div`s or `span`s to adjust for different layouts is cumbersome when it comes to code maintenance and readability. Lastly, we lose a great deal of our ability to reorganize our content in multiple ways when using multiple CSS styles. This means that rearranging the content in a table when we want to print or view the content in a different manner is more difficult, or would require a whole different page to accomplish.

To create a table, we first need to define its beginning and end with tags:

1. `<table></table>`

Next, we can define our heading row by adding our row tags (`tr`) and nesting dividing tags to represent our columns. Since we want items in this row to be recognized as headings, we use the `<th>` tag for table heading:

1. `<table>`
2. `<tr>`
3. `<th>ID</th><th>First Name</th><th>Last Name</th>`
4. `</tr>`

5. `</table>`

ID	First Name	Last Name
----	------------	-----------

**Dashed border lines in table examples depict table and cell edges that would not be visible without additional code or CSS.*

To add content to our table, we repeat the process using `<td>` (for table division) instead of `<th>` to represent a regular cell of data. Each repetition will add a new row of data to our table.

To build on our example, we will let the first column also be headings, which will represent each row. To do this we use `<th>` as the first set of tags in each row. We can add a few names as regular content using table division tags as well:

1. `<table>`
2. `<tr>`
3. `<th>ID</th><th>First Name</th><th>Last Name</th>`
4. `</tr>`
5. `<tr>`
6. `<th>1</th><td>John</td><td>Doe</td>`
7. `</tr>`
8. `<tr>`
9. `<th>2</th><td>Jane</td><td>Doe</td>`
10. `</tr>`
11. `</table>`

ID	First Name	Last Name
1	John	Doe
2	Jane	Doe

Spanning

Using tables to separate several smaller pieces of content horizontally within a layout element is still generally accepted, and is easier than styling divs, but it is still considered a less than ideal approach.

As we create more complicated table layouts, we may want to merge some of these fields together. We can create “extra-large” cells by adding `colspan` and `rowspan` attributes to the dividing tab (`th` or `td`). A `colspan` value of 2, for example, means the cell will fill two horizontal blocks (left to right) of the table. Likewise, a `rowspan` of two means the cell will fill two row’s worth of space in the column(s) it occupies, as in this example showing staffing for part of a week:

1. `<table>`
2. `<tr>`
3. `<th>ID</th><th>First Name</th><th>M</th><th>T</th><th>W</th><th>Th</th><th>F</th>`
4. `</tr>`
5. `<tr>`
6. `<th>1</th><td>John</td><td colspan="2">work</td><td></td><td rowspan="2">closed</td><td></td>`
7. `</tr>`
8. `<tr>`
9. `<th>2</th><td>Jane</td><td></td><td></td><td>work</td><td>work</td>`
10. `</tr>`

11. </table>

ID	First Name	M	T	W	T	F
1	John	work			closed	
2	Jane			work		work

Keep in mind that spans always move from left to right for columns and top to bottom for rows, starting with the cell you place the attribute in. Your values for the span must always be positive, and larger than zero.

While these examples outline where table cell edges are with dotted lines, we can selectively enable these borders through cell attributes and/or through CSS styling. The best approach is to keep these visual changes in CSS, preserving the structure/style separation of duties between our HTML and CSS, making future visual changes easier to maintain.

This page titled [1.7: Tables](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.7: Tables](#) by [Michael Mendez](#) has no license indicated.

1.8: Forms

Forms

Forms drive the internet. They are perhaps the most critical element in creating an interactive experience for your end users, and allow you to take in input. Forms define places on a page where the user's interaction can add, change, interact with, or remove the data in your system. The actions and fields you allow in your form determine what the user is allowed to do, and what information he is allowed to see.

Form elements range from username and password style boxes to large text fields, drop down lists, checkboxes and more. All of the elements within a form block are sent from that page to the destination attribute of the form declaration, called an action. When the user hits send this information is then made available in one of several ways to the receiving page or script.

To create a form section, we provide the form with a name, id, action, and method. An example with blank attributes looks like this:

1. `<form name="" id="" action="" method=""></form>`

Our form's name and id are how we will refer to it in our code when interacting with it using CSS, JavaScript, or other languages. The action is where the page should send the information (and where the browser will go when we hit send). Our method is how we will send our information, using either GET or POST.

Get

Sending the data using the get method places all of the form fields by name and value (called a key and value pair) into the address bar, making our URI longer by appending each item to the receiving page's address. An easy way to remember this is that the user "gets" to see the information that was sent, as it will appear in the address bar at the top of the browser. The benefit of using the get method is that the destination can be bookmarked with the data that was sent. So, if your form is used to search a library and filter results, you could save the result as a bookmark, and return to the page in the future, seeing the same results without filling out the form again.

While beneficial, there are two instances in which we DO NOT want to use get: either we do not want the user (or anyone) to see what was sent, such as passwords or confidential information, or we want to send a lot of information. There is a practical limit to how much data can reliably and safely be passed using get, although no formal ceiling. The practical limits are those created by the browser or server's ability to store the information being sent. When developing large get requests, determine which browsers you want to support, and how old, and figure out which of the oldest has the lowest maximum threshold.

Post

Posted data is sent from the browser to the server in the background, as the client and server first begin to talk. The data is sent in the headers (see) of the communication, and are not visible to the end user. Pages bookmarked with the post method will not have access to the information later on, and that information is lost if the user leaves the page.

How the data is used and values or new content returned bring us to scripting languages. Skip to the server-side language section of this text to learn about that process.

Form Fields

When a webpage with a form is rendered, we can identify a specific field for the user to start with. You may have experienced this in action when you load a website and find the cursor already in a textbox. This is autofocus. To include this function, simply add the attribute autofocus to the field the user will want or need first. We can also apply placeholders to our text fields that tell the user what we want them to enter with the placeholder attribute. To begin, we will add a text input inside our form tags for a name field:

1. `<form name="" id="" action="" method="">`
2. `<input type="text" placeholder="Your First Name" autofocus name="name" />`
3. `</form>`

Many of the new elements of HTML5 we look at will also assist us with our validation tasks as users fill out forms. These inputs will attempt to validate and/or limit user entry to only valid data. By doing this immediately, we create a better experience for both

the user and the programmer. Traditionally, validation had to be done when data was sent to the server, resulting in the page reloading if there were errors. The other popular approach was to perform validation using JavaScript on the client-side (avoiding the reload), but validation would still have to be repeated on the server in the event the end user had JavaScript disabled. Some of the more useful input types are the following:

`<input type="url">` Will attempt to format the user's text into a proper link, or display an error. `<input type="email">` Will make sure an email entered is in proper format, or display an error.

We can also create an input that limits values to a fixed range and increment limitations, which we used to have to display to the user on the page, and then validate after entry:

1. `<input type="range" min="10" max="50" step="5" value="30">`

These limits on a range (shown as a slider) also are valid on a number field as well (shown as arrows):

1. `<input type="number" min="10" max="50" step="5" value="30">`

HTML5 also introduces a wealth of calendar and time controls. We can specify a date, week, or month as well as a time, day and time, and local day and time. Each of these fields will limit the user's entry to valid fields for that type.

Calendar options:

1. `<input type="date" name="date"/>`
2. `<input type="week" name="week"/>`
3. `<input type="month" name="month"/>`

Time Options:

1. `<input type="time" name="time"/>`
2. `<input type="datetime" name="dateTime"/>`
3. `<input type="datetime-local" name="localDateTime"/>`

Learn more

Keywords, search terms: Tables, forms

Do not Use Tables For Layout: <http://webdesign.about.com/od/layout/a/aa111102a.htm>

Mozilla HTML Forms Guide <https://developer.mozilla.org/en-US/...ide/HTML/Forms>

This page titled [1.8: Forms](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.8: Forms](#) by [Michael Mendez](#) has no license indicated.

1.9: Canvas

The canvas element (new as of HTML5) allows us to approach pages with greater control by drawing and creating SVG-style graphics on the page in real time with JavaScript, and giving us the ability to animate and control the motion of our elements. With these new abilities, it is now possible to create browser games and highly interactive pages without the use of flash, additional components, or even pre-existing images (not that this would be a best approach in every situation).

Terminology and integrated functions are focused around the concept of art and media graphics, including functions like `stroke()` and `fill()` among others, that expedite your ability to create an image on the screen without a verbose background in graphic arts and mathematical modeling.

Each of the items we create can become an object of its own, and can be grouped with multiple layers or elements as one item. Supporting browsers can understand an object's dimensions and relationship to other elements, bringing native drag-and-drop into play. Page elements that support drag-and-drop can add the `draggable` attribute to their declaration.

Calling this element a canvas is intentional, and conveys an accurate portrayal of how to treat it. When we create a set of canvas tags and set our width and height, we have effectively “hung” a blank painting on the “wall” of our web page. In our examples below, we will be using a number of values to determine where things we “paint” on the canvas will be.

This is done by using value pairs, or X-Y coordinates. The top-most left-hand corner of the canvas is always (0, 0)—0 pixels to the right, 0 pixels down. This is different from a graph where 0, 0 is in the middle of the page. Our values for X and Y will move our drawing point to the right and down as they grow larger.

In our first example below our canvas size is 300x300, which means the bottom-most right-hand point is (300, 300). Any values larger than this, or points with negative values, will move part or all of our drawing off of our canvas.

Rectangles

We will get right into canvas, since it is a visual process, and can be a lot of fun. To begin, we need to create a canvas element on our page:

1. `<canvas height="300" width="300"> </canvas>`

While our canvas is still empty what we have done is allocated a space (just like sizing a `div`) to declare what part of the page belongs to our `div`. The width and height tags we provided are required from the start, and since we have not defined an offset or placed the canvas in another container, it will start from the top left corner of our page, again just like a `div`.

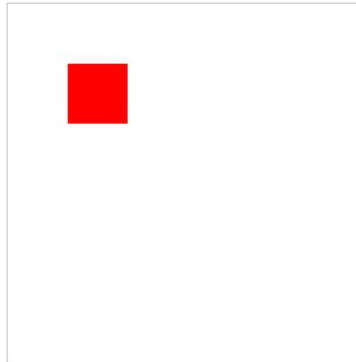
The act of drawing on our canvas is a several-step process. We have to declare how we want our element to appear (for example the `fillStyle` or `strokeStyle` attribute), where we want it to start from, and what type of line or shape we intend to create (for example, `fillRect` or `strokeTriangle`).

When we declare a shape, we need to convey its size and location. For a rectangle, we do this by setting its starting location (top left corner) as X and Y values, and then by adding its width and height. To add a solid rectangle to our canvas, we will have to add some JavaScript to our page. Since we have not reached JavaScript yet in this text, do not worry if you do not understand every little bit—we will get there. For now, focus on understanding which position of the arguments is used for different settings.

In the header of our page, we need to add the following JavaScript code, identifying what element we want it to affect, and what we want the drawing to be:

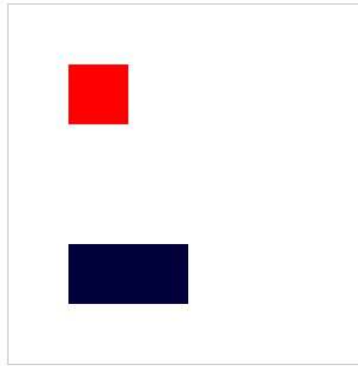
1. `<!DOCTYPE html>`
2. `<html>`
3. `<body>`
4. `<canvas id="canvas" width="300" height="300" style="border: 1px solid #c3c3c3;">`
5. Oh no! This browser does not support HTML5 :(
6. `</canvas>`
7. `<script>`
8. `var canvas=document.getElementById("canvas");`
9. `var canvas1=canvas.getContext("2d");`

```
10. canvas1.fillStyle="#FF0000";
11. canvas1.fillRect(50, 50, 50, 50);
12. </script>
13. </body>
14. </html>
```



Saving and refreshing our screen should now give you a single lonely rectangle set a little inside the top left corner of your page. You may be wondering about the “Oh no!” line of our example. When our page loads, the “Oh no!” content is placed on our page. When our canvas layers are rendered, this content is then covered up. If HTML5 is not supported (or JavaScript is disabled) our canvas is not drawn, leaving the original text which we can use to tell the user something is wrong. In our fill style declaration we used a color reference as a hex value. We can also use a standard color word like red, or use a function call that takes a red, green, blue, and opacity value set to generate a color. In our fillRect declaration we defined the starting position from the left and top, as well as its width and height, respectively as fillRect(left, top, width, height). In our initial example, all values were 50. Let us add a second rectangle that is wider than it is tall, and move it much further down our page:

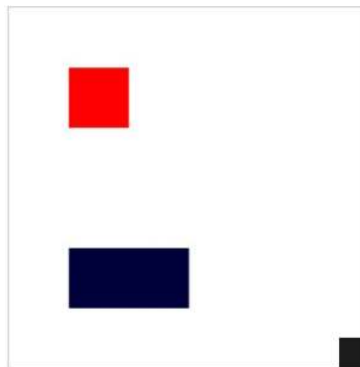
```
1. <!DOCTYPE html>
2. <html>
3. <body>
4. <canvas id="canvas" width="300" height="300" style="border: 1px solid #c3c3c3;">
5. Oh no! This browser does not support HTML5 :(
6. </canvas>
7. <script>
8. var canvas=document.getElementById("canvas");
9. var canvas1=canvas.getContext("2d");
10. canvas1.fillStyle="#FF0000";
11. canvas1.fillRect(50, 50, 50, 50);
12. canvas1.fillStyle= "rgba(0, 0, 50, 100)";
13. canvas1.fillRect(50, 200, 100, 50);
14. </script>
15. </body>
16. </html>
```



You will notice that even though we now have two blocks, we do not need to change the names we used when declaring our second rectangle from our first. This is because we are setting values, then calling a function to draw the element, and are not storing the values in our script as objects in our JavaScript code.

Keep in mind our drawings can be declared in a manner that draws them outside or extending beyond the confines of our canvas. They will technically be drawn, however the canvas will not expand to meet the needs of your drawing. To see this in action, we can add one more rectangle the same size as our first, but place it so it overdraws our canvas, so we only see a portion of it:

```
1. <!DOCTYPE html>
2. <html>
3. <body>
4. <canvas id="canvas" width="300" height="300" style="border: 1px solid #c3c3c3;">
5. Oh no! This browser does not support HTML5 :(
6. </canvas>
7. <script>
8. var canvas=document.getElementById("canvas");
9. var canvas1=canvas.getContext("2d");
10. canvas1.fillStyle="#FF0000";
11. canvas1.fillRect(50, 50, 50, 50);
12. canvas1.fillStyle= "rgba(0, 0, 50, 100)";
13. canvas1.fillRect(50, 200, 100, 50);
14. canvas1.fillStyle = "rgba(20, 20, 20, 20)";
15. canvas1.fillRect(275, 275, 50, 50);
16. </script>
17. </body>
18. </html>
```



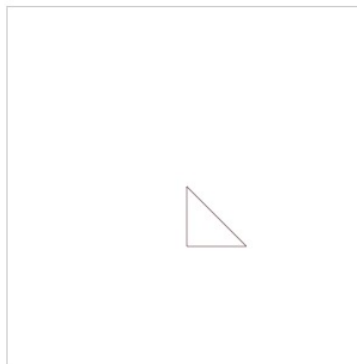
Additional Notes

Keep in mind that JavaScript, which we are getting a sneak preview of, is case sensitive. This means Canvas1 is considered different than canvas1!

Even though this last rectangle is still 50 by 50, we only see the 25x25 of it that fits inside our canvas dimensions. Now that we have played with rectangles, we will replace them with triangles. We will also just give them borders without a fill color. To do this, we will define the line segments that make up our triangle, and use `strokeStyle()` instead of `fillStroke()`:

Triangles

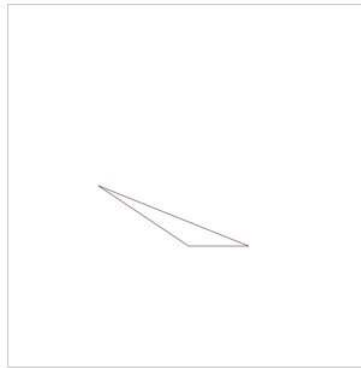
```
1. <!DOCTYPE html>
2. <html>
3. <body>
4. <canvas id="canvas" width="300" height="300" style="border:1px solid #c3c3c3;"> Oh no! This browser does not support
   HTML5 :(
5. </canvas>
6. <script>
7. var canvas=document.getElementById("canvas");
8. var canvas1=canvas.getContext("2d");
9. canvas1.beginPath(); //declare the beginning of settings for our line
10. canvas1.strokeStyle = "rgba(50, 0, 0, 0.5)";
11. canvas1.moveTo(150,150); // set the starting point of our "pen" to the middle
12. canvas1.lineTo(150,200); // move our "pen" down 50 pixels, drawing a line
13. canvas1.lineTo(200,200); // move our "pen" 50 pixels to the right
14. canvas1.closePath(); // Draw a direct line back to our starting point
15. canvas1.stroke(); // Visually place the defined line on the page
16. </script>
17. </body>
18. </html>
```



Saving and refreshing should now remove the rectangles we drew earlier and replace them with a right angle triangle positioned with the right angle in the middle of the canvas. By adjusting our X and Y values in `moveTo` and `lineTo` variables, we can move our triangle around the page. We will change just one point (our starting point) and see how different our triangle looks:

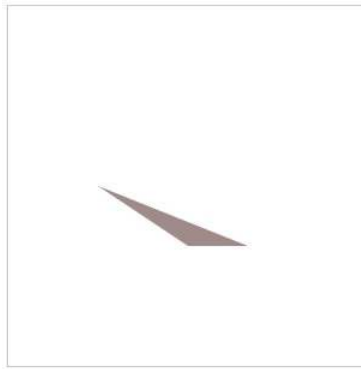
```
1. <!DOCTYPE html>
2. <html>
3. <body>
4. <canvas id="canvas" width="300" height="300" style="border: 1px solid #c3c3c3;"> Oh no! This browser does not support
   HTML5 :(
5. </canvas>
6. <script>
7. var canvas=document.getElementById("canvas");
```

```
8. var canvas1=canvas.getContext("2d");
9. canvas1.beginPath(); //declare the beginning of settings for our line
10. canvas1.strokeStyle = "rgba(50, 0, 0, 0.5)";
11. canvas1.moveTo(75,150); // set the starting point of our "pen" to the middle
12. canvas1.lineTo(150,200); // move our "pen" down 50 pixels, drawing a line
13. canvas1.lineTo(200,200); // move our "pen" 50 pixels to the right
14. canvas1.closePath(); // Draw a direct line back to our starting point
15. canvas1.stroke(); // Visually place the defined line on the page
16. </script>
17. </body>
18. </html>
```



To convert our outlined triangle to a filled, solid object we just need to convert our stroke settings back to fill:

```
1. <html>
2. <body>
3. <canvas id="canvas" width="300" height="300" style="border: 1px solid #c3c3c3;"> Oh no! This browser does not support
   HTML5 :(
4. </canvas>
5. <script>
6. var canvas=document.getElementById("canvas");
7. var canvas1=canvas.getContext("2d");
8. canvas1.beginPath(); //declare the beginning of settings for our line
9. canvas1.fillStyle = "rgba(50, 0, 0, 0.5)";
10. canvas1.moveTo(75,150); // set the starting point of our "pen" to the middle
11. canvas1.lineTo(150,200); // move our "pen" down 50 pixels, drawing a line
12. canvas1.lineTo(200,200); // move our "pen" 50 pixels to the right
13. canvas1.closePath(); // Draw a direct line back to our starting point
14. canvas1.fill(); // Visually place the defined line on the page
15. </script>
16. </body>
17. </html>
```



We can move beyond straight lines in order to draw other shapes by using Bezier curves, quadratic curves, and arcs. Each of these allow us to define different points on our lines and to curve our line between those points. For now, we will look at Bezier as an example of how to approach multi-point curves. Bezier lines allow for two control points as opposed to the one allowed in quadratic curve, so the programmatic difference is essentially just one less point defined for a quadratic than a Bezier (limiting the shape your line can take).

Bezier Curve

```
1. <html>
2. <body>
3. <canvas id="canvas" width="300" height="300" style="border: 1px solid #c3c3c3;"> Oh no! This browser does not support
   HTML5 :(
4. </canvas>
5. <script>
6. context.beginPath();
7. context.moveTo(10, 130);
8. context.bezierCurveTo(0, 10, 290, 10, 290, 290);
9. context.lineWidth = 10;
10. context.strokeStyle = "black";
11. context.stroke();
12. </script>
13. </body>
14. </html>
```

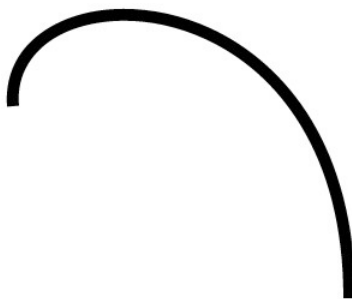


Figure 1.9.6
PageIndex

{1}: Copy and Paste Caption here. (Copyright; author via source)

By setting a `moveTo` point, we are starting our curve from that point just as with a regular line. From here, our Bezier function call takes three sets of points, our two control points, and end point. As a challenge, try adjusting your Bezier points to turn this example into a drop, or marker style, symbol.

Drawn Text

We can even draw text on the screen. While this might seem redundant as we have done that since we began this section, drawing text on a canvas can help us complete logos or draw letters without specifying every line needed to create the letter itself. This means the text becomes part of the canvas and cannot be copy/pasted. To create text, we simply need to define the style, string, and starting location. Replacing our earlier examples, our new canvas1 definition turns into:

1. `<script>`
2. `var canvas=document.getElementById("canvas");`
3. `var canvas1=canvas.getContext("2d");`
4. `</script>`

The benefit of creating text as a layer on a canvas instead of styling it with CSS allows us additional mechanisms to manipulate our words.

With this set of techniques, we can now recreate icons and logos without needing actual image files, and can scale them to fit as our page size and layout changes. This also makes it more difficult for your drawn images to be “borrowed.” As there is no image file to save, it becomes more difficult (though not impossible) for anyone trying to use your creations. Let us add text to this canvas by setting up a splash of color for our text, then defining our font, text and start location:

1. `var gradient=ctx.createLinearGradient(0,0,c.width,0);`
2. `gradient.addColorStop("0","red");`
3. `gradient.addColorStop("0.5","yellow");`
4. `gradient.addColorStop("1.0","blue");`
5. `ctx.fillStyle=gradient;`
6. `ctx.fillText("Let's see some color!",10,90);+`

Learn more

Keywords, search terms: Canvas, css graphics

See what Canvas can do! <http://net.tutsplus.com/articles/web-roundups/21-ridiculously-impressive-html5-canvas-experiments/>

The full specifications <http://www.w3.org/TR/2009/WD-html5-20090825/the-canvas-element.html>

This page titled [1.9: Canvas](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.9: Canvas](#) by [Michael Mendez](#) has no license indicated.

1.10: Media Support

Video

The abundance of audio and video material on the internet grew over time as bandwidth improved and storage space became cheaper and ubiquitous. Through this time, many approaches were brought forward to facilitate enjoying the material. Software, file extensions, browser add-ons and more attempted to fill the void. Now, HTML5 has added both audio and video tags to make it easier to integrate media into our pages. These tags make embedding media easier, but (so far) are limited in the number of file formats they support.

Additional Notes

You can convert video files to OGG with any supporting software just as you would convert to any other video format.

Our new video tag supports a number of features through attributes like automatically playing (autoplay), looping the file (looping), on screen controls (controls), preloading the video before it is played (preload), error handling (onerror), and even displaying an image when the file is not in use (poster) and also includes width, height, and source attributes of the file.

The video files we wish to use must be in the OGG format, which is an open source video format. We define our video very similarly to other elements we take into our page. An initial video that simply plays when the page is loaded can be completed by adding the following:

1. `<video src="ourfile.ogg" width="200" height="150" autoplay>`

Take note that this is a pretty rude solution in many cases, as the file will start to play as soon as enough of it is ready, and the user is left with no means of control except closing the page entirely (think about the lovely ads we have all been subject to that use this type of technique). To address this, we will add some more attributes:

1. `<video src="ourfile.ogg" width="200" height="150" autoplay controls preload="auto" poster="videoImage.png">`



Now we see a splash screen and controls. Our other options for preload (auto lets the browser decide) are none and metadata. None will stop all preloading activity, while metadata loads information about the video like length and dimensions. Audio files are actually identical to this approach (even the use of .ogg files) with exception to the ability to define the width and height. All we need to change is our vocabulary, using the word audio instead of video in our tag.

Audio

- `<audio src="ourAudio.ogg" autoplay controls preload="auto">`



The use of audio in a site can add a number of effects, from background music (be kind—allow the user to stop it if they want), to little noises that aid in navigation (these might interfere with screen readers) to embedded players that allow the user to select and play a particular audio file.

Like image formats, audio files can be encoded in a variety of formats that provide different levels of quality. The two formats most commonly found are .mp3 and .wav, standing for motion picture extract group (3rd set of standards) and wave, respectively. The difference lies in their method of compressing and storing the information needed to reproduce the sound. MP3 files are more compressed, meaning they are smaller and require less time and space to transmit, but contain less information and as such have a lower audio quality. Wave files, on the other hand, are typically much larger and sound better. This means they are more taxing on a website as they require more storage space and will take longer to load.

Which format you use will obviously depend on what is more important to you—speed of delivery and space, or quality of sound for the user. Some things to take into consideration are that most users report hearing little difference in quality between the formats, especially when the file is played on a mobile device or computer with basic entry-level speakers attached. If you have ever purchased a digital album or single online, you may have been presented with the option between these or other formats to accommodate the more discerning audiophiles among us.

Learn more

Keywords, search terms: Audio, video, embed(ded) audio video

Example of commercial htm5 based tools: <http://flowplayer.org/>

Example of flash based tools: <http://www.tubesnack.com/>

This page titled [1.10: Media Support](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.10: Media Support](#) by [Michael Mendez](#) has no license indicated.

1.11: Mobile Device Support

When you are creating your HTML pages and following the responsive design patterns that restrict tags to structural as much as possible, you have already laid the groundwork to support mobile devices.

When all of your page content is broken down into logical pieces, and those pieces have ID and/or class attributes, we will be able to show, hide, or reposition those elements using CSS. When we determine the screen dimensions the user has on their device, we can decide what or how to show content from our regular page. The key to this, of course, are the ID and class tags. Anything that you may want or need to control (move, alter, edit, etc.) as a single piece by itself should have an ID tag. Elements that will share the same changes (for example, pictures, links, paragraphs, etc.) should have the same class or classes so they can be changed at the same time.

A single element, say a paragraph in a group of paragraphs, can have both an ID and a class, and can receive style changes from CSS due to both. We will get to this in more depth in CSS, but keep it in mind as you build pages now so they are ready for you later.

If you are reading this chapter looking for examples on HTML5 support of things like [Geolocation](#), these will be found in the JavaScript section as they require the use of a scripting language to function, and are not available through HTML markup.

Important considerations for mobile development go beyond styling changes and begin to include performance measures as well, since mobile devices often lack computing power that can match a desktop or laptop's capacity. Skipping some of your fancier animations or large background images to trim down your loading times is another responsive-style adjustment to your site alongside other CSS changes. See the resources below for some examples of methods that may help your site improve response times.

Learn more

Keywords, search terms: Mobile optimization, mobile performance

Optimization tips and tools <http://www.html5rocks.com/en/mobile/>

Compatibility Charting: <http://mobilehtml5.org/>

Test your browser(s): <http://html5test.com>

Follow new developments: <http://www.mobilexweb.com/>

This page titled [1.11: Mobile Device Support](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.11: Mobile Device Support](#) by [Michael Mendez](#) has no license indicated.

1.12: Tags to Avoid

Now that we have focused on the new features of HTML5 and warned about the inefficient methods developers resorted to in the past, let us take a look at a few other tags and methods to avoid using in our code. While some of these are already deprecated, not all are, and they all take away from our goals of separating structure from style and responsive design.

**, <i>, **

All of these place emphasis on text, and can be replaced with CSS. Using these within your code means a change to your CSS style will not necessarily determine whether or not the style of text is affected for content wrapped in these tags. It also means you have to edit style in two places, both your CSS and HTML files.

Again, we can control font (with even more control) using CSS, and hard coding your font into your page can override what you wanted in your CSS.

**
**

I know, I have (and will) use breaks in my examples. What I mean to discourage here is using breaks as a quick fix when you should be using `<p>` or something more appropriate. If another tag better identifies your content, but you do not want some of the automatic styling that comes with the tag, we can override those attributes in our CSS, so use the best tag for your content. We will see how to adjust it to your needs when we look at CSS later.

Styling Attributes

Just like not using tags that style in our HTML, we should avoid using attributes that affect page appearance as well, like aligning content, or applying borders, colors, and other CSS elements.

This page titled [1.12: Tags to Avoid](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.12: Tags to Avoid](#) by [Michael Mendez](#) has no license indicated.

1.13: Rule Structure

CSS

Selectors

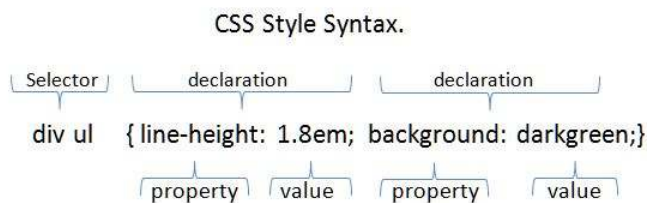


Figure 1.13.1: CSS Rule Structure. (By [Armchair](#) [Public domain], via [Wikimedia Commons](#))

The selector is a good place to start with CSS. It is used to identify which item(s) the rule(s) following it will apply to, and is the first element of a rule definition. A selector can refer to an existing tag that is part of the HTML structure like unordered lists, paragraphs, inputs, or it can refer to a custom class, element's name, or ID that we create. Selectors are followed by a set of braces { }, and the rules we enter in between the braces (the property we want to adjust, and the value we want it to have) apply to the selector we specify. We will look at some of these here (there are 40+ as of initial CSS3 components), focusing on those that are more frequently used. Take note that in CSS, we use the colon (:) instead of an equal sign (=) to assign values.

Classes

To take a look at how we can attach a “where” concept to our rules, we will attach some classes to our tags. This allows us to apply different styles to the same selector type, while adding the ability to be more specific about which occurrence we are talking about. We use classes by giving them rules in CSS and referring to them as attributes in elements that support them from our HTML document by using `class=“nameOfOurClass.”` When defining classes in our CSS file, we precede their name with a period to specify they are a class.

IDs

The ID attribute is used similar to how a class is used, with two differences:

- The ID is defined with a # instead of a period, e.g., #warning.
- You should use an ID only once on a page, as it should identify something unique to the document.

We already have our ID references in our HTML file anywhere we used `id=“ourNameHere”` attributes in our tags.

Examples

We will begin with a simple example and adjust our unordered list by disabling bullets. For now, we will put the rule in our <head> nested within <style> tags, which will allow the rule to apply anywhere on the page where the selector (in this case, ul) exists:

1. <style>
2. ul {
3. list-style-type: none;
4. }
5. </style>

We could also write this rule in a more condensed form:

1. <style>
2. ul {list-style-type: none;}
3. </style>

Since CSS rule sheets ignore white space, the line breaks only serve to make the document more readable when we interact with it. Minifying it (removing all of the extra line breaks by hand or with the use of a script) can reduce the size of larger style sheets, which can help performance (especially in mobile devices where we know this is of more concern).

By adding the rule above, we dictate the appearance of all unordered lists on our page. Rules specify the “What, Where, and When” aspects we want to define for the given selector. The “What” is the actual change we want to see—text in a certain color, an image in a specific location, and so on, just like our example above. In this case, we wanted to hide the default bullet markers. The “Where” is one or more specific locations, defined as a particular named element or when certain conditions are met, i.e. before every paragraph (`p.before{}`), or every link on the page (`a{}`). Finally, the “When” aspects allow us to control the site’s appearance based on different factors like if the display is mobile sized, or if the user clicked the print button (`@media print {}`).

By placing these examples in the `<head>` of our page, we have created an internal style sheet. This denotes that all of the style information we need to reference can be found in the page itself. We could also create external files with our rules. This allows us to have one or more CSS files in our site, and to apply those rules to one or more pages. The external file method is what we will predominantly use as it best fits responsive design, and separates the tasks our files are targeting.

Important Reminder

Using the inline method outside of JavaScript violates our responsive design and separation of duties! Another downside to this method is it would have to be applied to every paragraph we wanted it on within our page.

To connect a CSS file to our webpage, we would move all of our rules (everything inside the `<style>` tags) into a new document. After saving this new file with the `.css` extension, the browser will understand that everything contained is to be treated as if it were in our `<style>` tags. We can connect it to our page much like linking to another page in our server:

1. `<link rel="style sheet" type="text/css" href="mystyle.css">`

In our `<link>` tag we provide a `rel` attribute of style sheet to identify the file’s purpose. The `type` attribute tells the browser how to expect the contents to be formatted (“text/css”), and `href` provides the location of the file. If the file was not in the same folder as the web page we are using, we would adjust the link just as we would in an `<a>` tag.

Past practice has been to create entirely different sites, complete with customized CSS and HTML pages, to support mobile devices. This kicked off the `m.yoursite.com`, or `mobile.yoursite.com` period where heavily modified versions of sites were maintained separate from the primary, desktop version. Now, with the use of responsive design methods and the newer features of CSS3, we can use one master CSS file for all aspects of our site.

You may wish to separate your CSS files while you are working on them, as one master file can become tedious to search through. It may also help to add multi-line or long comments in your file to delineate between organized sections of your CSS. Ultimately, by your release date, you will want to get all of your CSS back into one file. Compressing it with tools like <http://www.csscompressor.com/> which remove the whitespace that makes the file readable while developing, and reduces the file size making transmission faster for the user.

Our final method of applying rules is inline. This refers to the practice of inserting the rule inside the attribute tag of the item in question. If we wanted to change the text color of a particular paragraph, we could go right to our HTML file and alter our regular `<p>` tag to read:

1. `<p style="color:red;">Our now RED paragraph!</p>`

Here we do not need our selector as it is the tag itself, and we also do not need our braces, or quotes within the rules. All we include in our attribute’s value are the surrounding quotes and the rules we want to apply, still using `:` and `;` appropriately. This is an effective means of responding to an event, or changing appearance through JavaScript.

There are a number of other things we can do with selectors in order to identify what we want to interact with. The `*` character can be used to apply to everything within page, or within a selector if one is specified. For example, `* { color: blue }` would make all text on the page blue (assuming it is the last rule applied... we will get to that) while `footer * { color: blue }` would make all items in our footer blue. This can be handy for development and testing to quickly adjust a number of items, but should be avoided in production as processing the rule can require examining a large number of elements, potentially increasing page load times.

We can target specific items by referring to them using their ID. To do this, we put a `#` in front of their ID name. As an example, we will create two divs, one with an ID of `todo` and another with an ID of `complete`, to make a quick task list:

1. `<div id="todo">`
2. `<ul>`

3. Check the mail
4. Go to the store
5. Take the dog to the vet
6.
7. </div>
8. <div id="completed">
9.
10. Rent a movie
11. Make grocery list
12. Make vet appointment
13.
14. </div>

- Check the mail
- Go to the store
- Take the dog to the vet
- Rent a movie
- Make grocery list
- Make vet appointment

By adding the following styles to this page, we can turn our To Do list red, and our Completed list green:

1. <style>
2. #todo { background-color:red; }
3. #completed { background-color: green; }
4. </style>

- Check the mail
- Go to the store
- Take the dog to the vet
- Rent a movie
- Make grocery list
- Make vet appointment

Important Reminder

Keep in mind we can only use an id name once on each page. If you want to apply these rules in more than one place, then you should use a class definition, coming up next!

We can use this technique on any tag that accepts an ID as an attribute (the actual rules we are allowed to use, however, differ depending on the element). To apply rules like this in multiple places, we will need to switch to a class definition. To do this, we simply change our # to a . and instead of relying on an element's ID, we give it a class attribute with a value of the selector name we create in our style definition. We will tweak our last example to split our To Do list into "today" and "tomorrow" lists and rename our completed div to "done." Then give both to do lists a class of todo, our done list a class of completed, and change our CSS rules to match:

1. <style>
2. .todo { background-color:red; }
3. .completed { background-color: green; }
4. </style>
5. <div id="today" class="todo">
6.
7. Check the mail
8. Go to the store

```
9. </ul>
10. </div>
11. <div id="tomorrow" class="todo">
12. <ul>
13. <li>Take the dog to the vet </li>
14. </ul>
15. </div>
16. <div id="done" class="completed">
17. <ul>
18. <li>Rent a movie</li>
19. <li>Make grocery list</li>
20. <li>Make vet appointment</li>
21. </ul>
22. </div>
```

- Check the mail
- Go to the store
- Take the dog to the vet
- Rent a movie
- Make grocery list
- Make vet appointment

To add even more emphasis to our completed list we can make the list inside it show text as crossed out. We could do this by applying a style to an ID we assign it, or by giving that ul a class, but we already have enough structure in place to specify what we want by using a descendant selector. This selector type is specified by saying we want elements “a” *inside* elements “b” to have the style applied. In our example, we want `<ul>`s inside a `.completed` crossed off, so we will add the following rule to our styles:

```
1. .completed ul { text-decoration: line-through; }
```

Now, if we add any more sections with a class of completed, they will also be crossed out automatically. You can try splitting your completed `<div>` into two pieces, like “yesterday” and “last week” to try it out.

Additional notes

Hover works on more than links! Just about any element has a hover state as long as the cursor position can be associated with it. Note: the z-index and position of layered elements can easily interfere with things like hover.

Hyperlinks are frequently styled so they change in appearance when they are hovered over or have already been clicked. We can assign our styles to these events by creating rules for `<a>` tags when they are unvisited, visited, or a hover (mouse pointer is over the link) state is attached:

```
1. a:link {
2. color:blue;
3. }
4. a:visited {
5. color:grey;
6. }
7. a:hover{
8. color:red;
9. font-weight:bold;
10. }
```

Try adding a link to your local Veterinarian’s office to the task in your todo list, and watch how it changes when you interact with it.

Sometimes you might find it easier to change something for all occurrences of an element that are not members of another rule. Instead of adding additional classes to those elements, we can specify an exception with the not state:

1. `div:not(.completed){ font-size:larger; }`

This rule will make text in any div (whether or not they have the `.todo` class) text larger. Another handy set of states are `:before` and `:after`. These selectors allow us to place and style content as a prepend or append action to our element. If we wanted to prepend a To Do: title to our `.todo` class elements, we could add the following rule:

1. `.todo:before{`
2. `content:"To Do:";`
3. `background-color:yellow;`
4. `color:red;`
5. `font-weight:bold;`
6. `}`

Important Reminder

Note: We cannot use the Content rule to add HTML to an element, only text. To achieve this, we would need to use JavaScript or another language that can manipulate the DOM.

The remainder of the selectors allow us to get even more specific, selecting the elements that appear before or after another given element, matching elements that have a particular attribute, or even matching attributes that contain a certain string as part of their value. For a full reference, I recommend the list at w3schools.com.² Our completed example should now look like this:

1. `<style>`
2. `.todo {`
3. `background-color:red;`
4. `}`
5. `.completed {`
6. `background-color: green;`
7. `}`
8. `.completed ul {`
9. `text-decoration: line-through;`
10. `}`
11. `div:not(.completed){`
12. `font-size:larger;`
13. `}`
14. `a:link {`
15. `color:blue;`
16. `}`
17. `a:visited {`
18. `color:grey;`
19. `}`
20. `a:hover{`
21. `color:red;`
22. `font-weight:bold;`
1. `.todo:before{`
2. `content:"To Do:";`
3. `background-color:yellow;`
4. `color:red;`
5. `font-weight:bold;`
6. `}`
7. `</style>`

```
8. <div id="today" class="todo">
9. <ul>
10. <li>Check the mail</li>
11. <li> Go to the store</li>
12. </ul>
13. </div>
14. <div id="tomorrow" class="todo">
15. <ul>
16. <li>Take the dog to the vet </li>
17. </ul>
18. </div>
19. <div id="yesterday" class="completed">
20. <ul>
21. <li>Rent a movie</li>
22. </ul>
23. </div>
24. <div id="lastWeek" class="completed">
25. <ul>
26. <li>Make grocery list</li>
27. <a href="http://www.dunkirkanimalclinic.com/"><li>Make vet appointment</li></a>
28. </ul>
29. </div>
```

To Do:

- Check the mail
- Go to the store

To Do:

- Take the dog to the vet
- Rent a movie
- Make grocery list
- Make vet appointment

Order of Precedence

There are some rules to our rules. There is a hierarchy to how they are applied in order to provide a semblance of order and avoid conflicts. First, we need to consider where the rule is located. In general, the “closer” the CSS file is to the line of code using it, the more likely that particular CSS file will override what is in the others. For example, rules in an external style sheet would be overridden by those in an internal style sheet. Just the same, styles in an internal style sheet would replace the previous style sheet included. Styles that are added to the page, or attributes that are inline with the element they apply to, become the “closest” to what it is targeting.

Useful Feature

JavaScript code that affects appearance will frequently apply inline modifications to apply style changes to a page, but will not affect the CSS file or other pages.

When we discuss the overriding nature, it is important to remember that this applies to the specific rule. A set of rules for an <a> tag defined in an external sheet will not necessarily be overridden by the <a> tag rules in an internal style sheet. Each rule within the definition is examined, and is still applied if that particular rules does not appear “closer” to the element. For example:

An External Sheet:

1. a {

2. color:blue;
3. text-decoration:none;
4. }

An Internal Sheet:

1. a { color:red;}

Our code:

1. <head>
2. <link rel="style sheet" type="text/css" href="mystyle.css">
3. </head>
4. <style>
5. a { color:red;}
6. </style>

Comparing the above example style sheets, we can see that the external sheet applies two rules: text color is blue, and there should not be an underline. However, our internal style sheet specifies a rule that color should be red. Since this rule is closer, our link will be red. The underline rule, however, is not defined in the internal sheet, so it will carry over and remain.

This also carries into the CSS rules themselves. Definitions can be general, like our example above that would apply to all links on our page, or they can be more specific to target a particular group or single link. The more specific we are, the greater influence our rule will have. For example:

1. <style>
2. a{
3. color:red;
4. }
5. div.block a{
6. color: blue;
7. }
8. </style>
9. <body>
10. Our RED link
11. <div class="block">
12. Our BLUE link

13. Our GREEN link
14. </div>

Above, our “a” style for our links is overridden in our div, as the block style is more particular, or close, to the actual link we want affected. However, our green link overrides our block style as it is inline, and applies to that specific link. From here, only a change in this inline style caused by JavaScript, or an “over-override” from a rule marked !important can trump the inline rule. The easiest way to debug CSS issues is with web developer tools that are an add-on or included in modern browsers like Chrome, that can show you exactly what rules are applied, overridden, and even let you test (temporary) changes before making them in your real file.

Caveat: User-enforced styles (settings the client enters in their browser, or applied by an accessibility program) also fit in to our precedence list, and can override any styles we set.

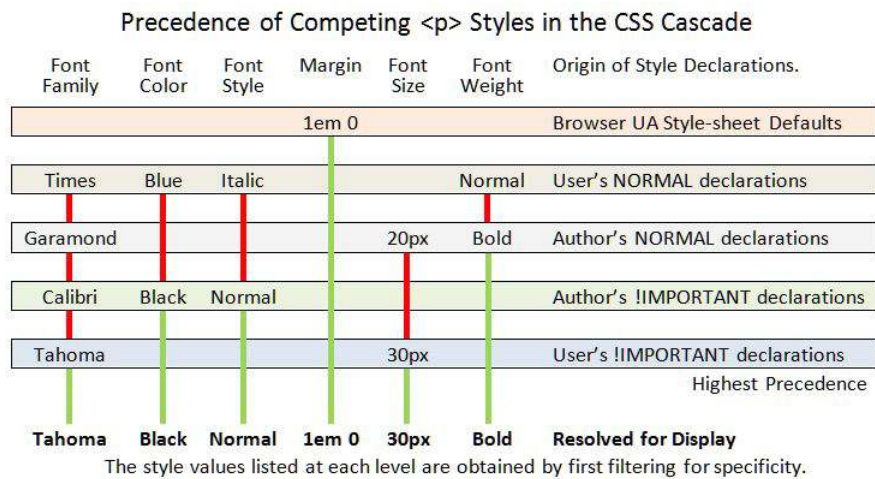


Figure 1.13.2: Document Markup. (By Armchair [Public domain], via Wikimedia Commons)

Learn more

Keywords, search terms: css rules, rule structure, selectors, css classes and ids

A list of all CSS properties: <http://www.bloobery.com/indexdot/css/propindex/all.htm>

All of the selectors: <http://www.w3.org/TR/CSS2/selector.html>

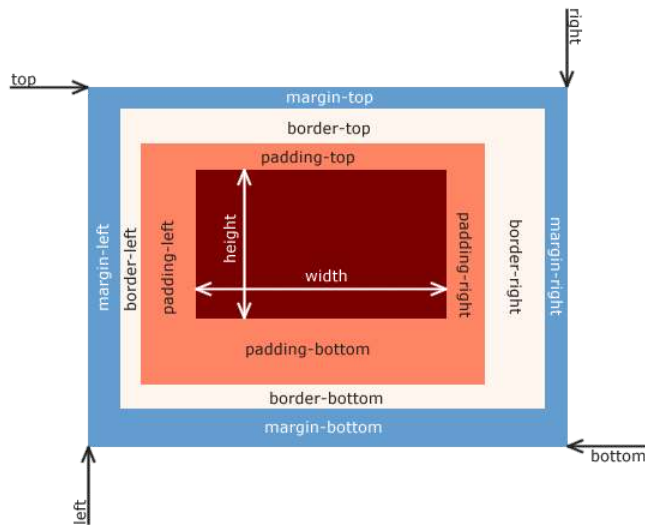
Some tips, tricks and more features: <http://www.instantshift.com/2010/03/15/47-css-tips-tricks-to-take-your-site-to-the-next-level/>

This page titled 1.13: Rule Structure is shared under a not declared license and was authored, remixed, and/or curated by Andrea Barrios.

- 2.13: Rule Structure by Michael Mendez has no license indicated.

1.14: Layout Formatting

Box Model



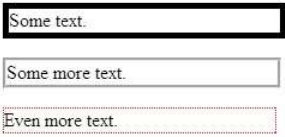
via Wikimedia Commons)

Figure 1.14.1: CSS Box Model. (By Matthias Apsel [CC0],

Borders

To better identify where our content falls on a page, and to signify that it is different from the material around it, we can adjust the borders on our elements. Borders can be enabled or disabled by the top, bottom, left and right of the element and can also have different styles like solid, double, grooved, dotted and dashed lines, among others.

To specify a full border, we simply use `border`, and can apply color, width, and style:

1. <code><style></code> 2. <code>p.one {</code> 3. <code>border-style:solid;</code> 4. <code>border-width:5px;</code> 5. <code>}</code> 6. <code>p.two {</code> 7. <code>border-style:groove;</code> 8. <code>border-width:medium;</code> 9. <code>}</code> 10. <code>p.three {</code> 11. <code>border-style:dotted;</code> 12. <code>border-width:1px;</code> 13. <code>border-color:red;</code> 14. <code>}</code> 15. <code></style></code> 16. <code><p class="one">Some text.</p></code> 17. <code><p class="two">Some more text.</p></code> 18. <code><p class="three">Even more text.</p></code>	 The image shows three text boxes stacked vertically. The top box has a thick, solid black border and contains the text "Some text.". The middle box has a medium-width, grooved grey border and contains the text "Some more text.". The bottom box has a thin, dotted red border and contains the text "Even more text.".
---	--

The full list of possible border styles is as follows:

Table 1.14.1 Border Styles

Value	Description
none	No border.
dotted	Dotted border.
dashed	Dashed border.
solid	Solid border.
double	Double (two lines) border.
groove	Grooved, 3d border.
ridge	Ridged, 3d border.
inset	Lowered (sunken) 3d border.
outset	Raised 3d border.
inherit	Inherits the same style as the parent element.

Margin and Padding

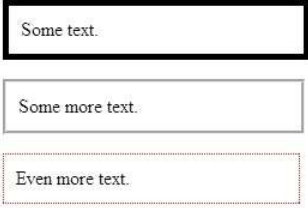
These related concepts allow you to control the amount of space between your content and its container, and between the container and objects around it. Padding controls the amount of space inside your container, for instance between text and a set of paragraph tags. You can remember padding as the inside by thinking about a padded room; the padding is only effective if it is on the inside of the walls.

By adding borders to our paragraphs as we did above, we can see the outline of where the paragraph fits into our page. Now, we will see the difference when we apply padding:

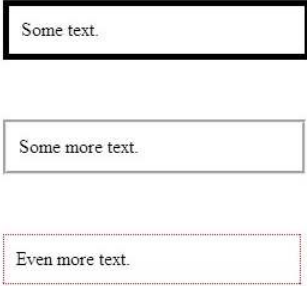
```

1. <style>
2. p.one {
3. border-style:solid;
4. border-width:5px;
5. }
6. p.two {
7. border-style:groove;
8. border-width:medium;
9. }
10. p.three {
11. border-style:dotted;
12. border-width:1px;
13. border-color:red;
14. }
15. p {padding:10px;}
16. </style>
17. <p class="one">Some text.</p>
18. <p class="two">Some more text.</p>
19. <p class="three">Even more text.</p>

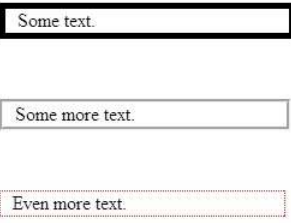
```



You will notice that the paragraphs still have not moved relative to one another, they each simply take up more space. In order to move them further away from each other, we can add a margin:

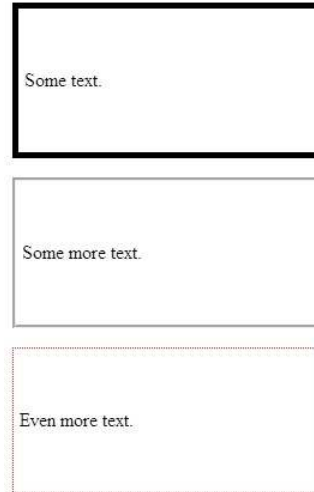
<pre>1. <style> 2. p.one { 3. border-style:solid; 4. border-width:5px; 5. } 6. p.two { 7. border-style:groove; 8. border-width:medium; 9. } 10. p.three { 11. border-style:dotted; 12. border-width:1px; 13. border-color:red; 14. } 15. p {padding:10px;} 16. p {margin:50px;} 17. </style> 18. <p class="one">Some text.</p> 19. <p class="two">Some more text.</p> 20. <p class="three">Even more text.</p></pre>	
---	--

With both examples, we can adjust our values by pixel or percent. We can also control the amount of change by each side of the object, by specifying top, bottom, left or right to our rules. To do this we need to edit our values to only pad the left side of our paragraphs, and only apply a margin to the bottom of each:

<pre>1. <style> 2. p.one { 3. border-style:solid; 4. border-width:5px; 5. } 6. p.two { 7. border-style:groove; 8. border-width:medium; 9. } 10. p.three{ 11. border-style:dotted; 12. border-width:1px; 13. border-color:red; 14. } 15. p {padding-left:10px;} 16. p {margin-bottom:50px;} 17. </style> 18. <p class="one">Some text.</p> 19. <p class="two">Some more text.</p> 20. <p class="three">Even more text.</p></pre>	
---	--

Instead of writing out multiple rules to adjust sides, we can combine them into one declaration by writing out our values clockwise, starting with top, as padding: top right bottom left or margin: top right bottom left, replacing the words with a fixed or relative value (they can be mixed) and by using zero as a place holder if we do not want the value changed from default:

```
1. <style>
2. p.one {
3. border-style:solid;
4. border-width:5px;
5. }
6. p.two {
7. border-style:groove;
8. border-width:medium;
9. }
10. p.three {
11. border-style:dotted;
12. border-width:1px;
13. border-color:red;
14. }
15. p {padding:50px 30px 50px 5px;}
16. p {margin:50px;}
17. </style>
18. <p class="one">Some text.</p>
19. <p class="two">Some more text.</p>
20. <p class="three">Even more text.</p>
```



Background

There is a lot we can do with the background of our pages. Colors and images can be applied to all or portions of our content, helping to highlight different elements of our site, and play a large part in the overall look and feel. We can specify colors by their

name if they are a basic color like red, white, blue etc. or we can provide its hex value, or the values for its red, green, and blue values.

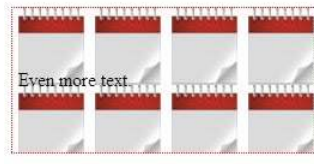
```
1. <style>
2. p.one {
3. border-style:solid;
4. border-width:5px;
5. Background-color: green;
6. }
7. p.two {
8. border-style:groove;
9. border-width:medium;
10. Background-color:#ff3355;
11. }
12. p.three {
13. border-style:dotted;
14. border-width:1px;
15. border-color:red;
16. background-color: rgb(33,66,99);
17. }
18. p {padding:50px 30px 50px 5px;}
19. p {margin:50px;}
20. </style>
21. <p class="one">Some text.</p>
22. <p class="two">Some more text.</p>
23. <p class="three">Even more text.</p>
```



To use images instead of colors, we can specify the image's location in our files, and can also dictate where we want to place it on our page, whether or not it should repeat, and whether it should move or remain in place when the user scrolls the page. By default, images will repeat to fill the space they are placed in. To prevent this, we can add a no-repeat attribute to our definitions. This time, we will use the background attribute as opposed to the background-color attribute. The benefit of this is that you can include both

in a set of rules on the same object (image first, color second), allowing you to have an image on top of a background color. Take note that in these examples, you will need to select your own images in place of those used below.

```
1. <style>
2. p.one {
3. border-style:solid;
4. border-width:5px;
5. background:url(clouds.jpg);
6. }
7. p.two {
8. border-style:groove;
9. border-width:medium;
10. background:url(calendar.jpg) no-repeat;
11. background-color:#ff3355;
12. }
13. p.three {
14. border-style:dotted;
15. border-width:1px;
16. border-color:red;
17. background:url(calendar.jpg);
18. }
19. p {padding:50px 30px 50px 5px;}
20. p {margin:50px;}
21. </style>
22. <p class="one">Some text.</p>
23. <p class="two">Some more text.</p>
24. <p class="three">Even more text.</p>
```



There are two ways of achieving this affect. The first is by using advanced styling through CSS using WebKit features supported by some browsers, and then adding style rules to create the effect as close as possible in other browsers. The second is by creating a repeatable gradient image. The first approach's reliance on WebKit provides support for Apple and Google products. For browsers

that do not use WebKit, we have to add extra rules to achieve the same effect. This is a more advanced example as it requires knowledge of each browser's needs to create:

```
1. <style>
2. #ourBackground {
3. background-color: #1a82f7;
4. background: url(ourFallbackImage.png);
5. background-repeat: repeat-x;
6. /* Safari 4-5, Chrome 1-9 */
7. background: -webkit-gradient(linear, 0% 0%, 0% 100%,
   from(#1a82f7), to(#2f2727));
8. /* Safari 5.1, Chrome 10+ */
9. background: -webkit-linear-gradient(top, #2f2727, #1a82f7);
10. /* Firefox 3.6+ */
11. background: -moz-linear-gradient(top, #2f2727, #1a82f7);
12. /* IE 10 */
13. background: -ms-linear-gradient(top, #2f2727, #1a82f7);
14. /* Opera 11.10+ */
15. background: -o-linear-gradient(top, #2f2727, #1a82f7);
16. }
17. </style>
18. <div id="ourBackground" width="300px" height="300px">
19. <br/>
20. Some <br/>
21. Text <br/>
22. Here <br/>
23. <br/>
24. </div>
```



1.

This code should produce an almost identical image in every browser, depending on which rule(s) the browser is able to execute:

The first three lines of this style script—

1. background-color: #1a82f7;
2. background: url(ourFallbackImage.png);
3. background-repeat: repeat-x;

demonstrate how we create the gradient effect through an image. In this example, ourFallbackImage.png would be a very skinny (1 pixel) wide image as tall as we want our gradient to be. By repeating this image across the X axis (moving horizontally) the image will fill the width of the parent object. By specifying the bottom-most pixel color from our image as the background, the gradient will appear to fill the page. The balance of our rules in this example achieve the same result through CSS, but also provides more control over the gradient without needing to create additional images.

Float

Floating an object with CSS allows us to move it around within its parent object, ignoring (to some extent) the other items near it. Note that float is only for left/right values, not top/bottom, even though their movement may feel that way as windows resize.

When multiple objects in the same container have the same float style, they will line up next to each other for as many as the container can fit. While this may sound confusing, we will look at it without the terminology: If you have a big box, and that box has small boxes in it, those boxes will fit as many of themselves left-to-right in a row as they can. Any boxes that do not fit will start a new “row” underneath.

The use of float is a big help to responsive styling. Boxes of content that normally fit side by side on a larger screen will automatically create more “rows,” with less items in each, to accommodate screens with less width. Create a page with the following code, and then play around with the size of your browser window to see the resizing in action:

```
1. <style>
2. .thumbnail {
3. float:left;
4. width:80px;
5. height:80px;
6. margin:5px;
7. }
8. </style>
9. <div>
10. 
11. 
12. 
13. 
14. 
15. 
16. 
17. 
18. 
19. 
20. </div>
```

Content before and after a floated element will attempt to wrap around it. When we do not want this to happen, we can add a rule to that element’s style to clear the floating effect. To do this, we would add `clear: left;` `clear: right;` or `clear:both` depending which sides we are concerned with.

Useful Feature

Since relative refers to moving the position from where it would be based on all of the other elements and rules, we can use negative values to “pull” an element in that particular direction.

Positioning

can specify, with great control, exactly where our elements are ultimately located in our window. CSS allows us to modify location to such an extent that an items position on the page can have no relation to its location in your code. Just as we have seen in many of our other rules, there are two methods to declaring position, fixed and relative. Here fixed elements specify the offset of pixels from a side or corner of the window, and relative declares that our values are moving the content from where it *would* have been if we had not changed it.

Here is how we might take a heading and force it to an offset from the top left corner as an absolute, meaning it will sit there no matter what else is above, underneath, or around it:

```
1. <style>
2. h2 {
3. position:absolute;
4. left:50px;
5. top:100px;
6. }
7. </style>
8. <h2>This is a heading with an absolute position</h2>
9. <p>
```

With absolute positioning, an element can be placed anywhere on a page. The heading below is placed 50px from the left of the page and 100px from the top of the page. As this text continues, you will see that the heading sits on top of the text as if it weren't even there. As this text continues, you will see that the heading sits on top of the text as if it weren't even there. As this text continues, you will see that the heading sits on top of the text as if it weren't even there. As this text continues, you will see that the heading sits on top of the text as if it weren't even there. As this text continues, you will see that the heading sits on top of the text as if it weren't even there. As this text continues, you will see that the heading sits on top of the text as if it weren't even there. As this text continues, you will see that the heading sits on top of the text as if it weren't even there. As this text continues, you will see that the heading sits on top of the text as if it weren't even there. As this text continues, you will see that the heading sits on top of the text as if it weren't even there.

This is a heading with an absolute position

<https://eng.libretexts.org/@go/page/96656>

critical items in a site. In our first example, we will see an image with a negative index that ensures it is behind our text. Then we will change our index value to make it higher, putting it on top of the text instead:

<pre>1. <style> 2. img { 3. position:absolute; 4. left:0px; 5. top:0px; 6. z-index:-1; 7. } 8. </style> 9. <h1>Here is some text</h1> 10. </pre>	
<pre>1. <style> 2. img { 3. position:absolute; 4. left:0px; 5. top:0px; 6. z-index:1; 7. } 8. </style> 9. <h1>Here is some text</h1> 10. </pre>	

Mouse Cursor

While this is not a regular feature in most sites, it can be an important player if you intend for your website to act as if it were an application.

We can add cursor rules to our selectors in order to change the appearance of the mouse cursor when that rule is active. Much the same as working in your operating system, we can select the regular icon, wait (also called working, busy, thinking, etc.), text insert, a pointer, a question mark, and a crosshair. While most of these have little use in the average web page, they come in handy when your end product is more application focused.

I would strongly recommend judicious use of cursor changes, and be sure that your changes are reverted back as soon as it is appropriate (i.e. change your waiting/busy back as soon as an event is complete) as forgetting to reset can leave your user thinking your site (or their system) is locked up or endlessly cycling. The full list of the available cursors is as follows:

Table 1.14.2 Pointer Styles

Value	Description
auto	(default) let the browser choose
crosshair	Crosshair, or “plus,” symbol
default	The default cursor
e-resize	Shows resize to the right (note all resize values are compass combinations)
help	The help (question mark) icon
move	Item can be moved
n-resize	Shows resize up
ne-resize	Shows resize up and right
nw-resize	Shows resize up and left
pointer	A pointer (arrow)
progress	The busy symbol (be careful with this one!)
s-resize	Shows resize down
se-resize	Shows resize down and right
sw-resize	Shows resize down and left
text	Text line (flashing or steady “I”)
w-resize	Shows resize left
wait	Shows busy, wait (be careful with this one!)
inherit	Inherits value from parent

Cursor styles can be applied when the element with a CSS attribute that affects the cursor is triggered. This is usually caused by hovering over the object, or when the user initiates an action, in reaction to which we apply the new style using JavaScript. Note that user triggered actions like busy icons normally need to stay “busy” until the script is done. In this case, the body tag should receive the attribute that affects the cursor so it continues to show as busy even if the user moves the mouse off of the button or other trigger that they used.

Learn more

Keywords, search terms: css layout, page formatting, positioning, css layers

LearnLayout: <http://learnlayout.com/toc.html>

Full layout example without tables: <http://www.w3.org/2002/03/csslayout-howto>

A set of basic layouts: <http://blog.html.it/layoutgala/>

This page titled [1.14: Layout Formatting](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.14: Layout Formatting](#) by [Michael Mendez](#) has no license indicated.

1.15: Font and Text Decoration

Color for Heading Tags

When we began our testing site, we started using HTML tags wherever we could to provide structure to our content with heading tags. We did this with the understanding that later we would redefine those tags so our headings looked how we wanted them to. That time has come. To start with some basics, we can use what we have already learned above by changing the color of the text and background for our heading tags:

```
1. <style>
2. h1{
3. color:red;
4. background-color:yellow;
5. }
6. </style>
7. <h1>This is an H1 heading</h1>
```

This is an H1 heading

We can also adjust our font family and size. You will notice that none of these changes affect anything outside of our headings. While you may see examples using the key terms for size, ranging from “extra extra small” (xx-small) to “extra extra large” (xx-large), it is a good idea to always be as specific as possible, as key terms can be treated differently between browsers. Instead, our examples will use percentage based and fixed font sizes. To do so, we will make our h1 content italicized and bring the size down to 20px tall:

```
1. <style>
2. h1{
3. color:red;
4. background-color:yellow;
5. font-style:italic;
6. font-size:20px;
7. }
8. </style>
9. <h1>This is an H1 heading</h1>
```

This is an H1 heading

These are just a few examples of the full power of font through CSS. Some “fancier” methods include effects like capitalizing, while simultaneously shrinking, your text (small-caps):

```
1. <style>
2. h1{
3. color:red;
4. background-color:yellow;
5. font-variant:small-caps;
6. }
7. </style>
8. <h1>This is an H1 heading</h1>
```

THIS IS AN H1 HEADING

Additional notes

Order is Important! Active style rules must come after hover rules, and hover must come after link and visited! Since a link being hovered over can already have been visited, and the active link can be the one with hover on it, this ensures the correct order of application of style.

Text Styles

While our next example seems like it applies more to font than text, a good way to remember what noun you want to use in your rule is whether the affect changes the way the letters appear or not. If they do, you probably want font. If not, then you probably want text as in these next examples.

First, we might want to add the lead spaces back into our paragraph's definition to make them appear more like a written document. We can also move our text around in our containing element by setting it to left (default), right, center, or stretch to fit with justify:

```
1. {  
2. text-indent:15px;  
3. text-align:justify;  
4. }  
5. </style>  
6. <p>
```

This is our paragraph for demonstrating some of the things we can do to text and font through the use of CSS. This is our paragraph for demonstrating some of the things we can do to text and font through the use of CSS. This is our paragraph for demonstrating some of the things we can do to text and font through the use of CSS.

This is our paragraph for demonstrating some of the things we can do to text and font through the use of CSS. This is our paragraph for demonstrating some of the things we can do to text and font through the use of CSS. This is our paragraph for demonstrating some of the things we can do to text and font through the use of CSS.

In addition to adjusting the font itself, we can decorate it with more affects like crossing it out, underlining it, or specifying that it should *not* be decorated, which is especially useful in eliminating the default lines under links:

```
1. <style>  
2. .strikeOut{text-decoration:line-through;}  
3. .titles{text-decoration:underline;}  
4. a {text-decoration:none;}  
5. </style>  
6. <span class="strikeOut">Text we want crossed out</span>  
  <br/>  
7. <span class="titles">Hitchiker's Guide to the Galaxy</span>  
  <br/>  
8. <span><a href="">A link with no underline</a></span>
```

~~Text we want crossed out~~
Hitchiker's Guide to the Galaxy
[A link with no underline](#)

Anchors

Following up on our ability to remove the underline from a link, there are some other special features we can control for our page anchors using CSS. By specifying link, visited, or hover in our link selector, we can control what happens before, during and after a link has been clicked. We can think of these like applying attributes in our HTML tags, except in CSS the special features are called pseudo-classes. Since we can specify any valid CSS rule, we can have our links change colors, alter the backgrounds, change text and font properties, and everything else we will look at. To see some of the basics in action, we will change our text colors for each action, and also our background color when we are hovering. Since you will need to interact with the links to see these in action, we will forgo an image here and you can test the code yourself:

```
1. <style>  
2. a:link {color:#FF0000; background-color:yellow;} /* unvisited link */  
3. a:visited {color:#00FF00; background-color:orange;} /* visited link */  
4. a:hover {color:#FF00FF; background-color:green;} /* mouse over link */  
5. a:active {color:#0000FF; background-color:white;} /* selected link */  
6. </style>  
7. <a href="" target="_blank">Here is our fake link!</a>
```

Visually Impaired Considerations

The Internet is obviously a highly visual medium, and it is good practice to keep in mind how your site will be consumed by those with differing visual needs. Here are a few techniques to be better prepared to serve a pleasant user experience.

First is text size. Modern browsers support increasing and decreasing page text on their own (by taking advantage of user style CSS properties). While this reduces the need to provide resizing style sheets for your users to select from, it does mean you should test increasing and decreasing the font size on your site from a browser (usually ctrl and +, ctrl and -). This will allow you to see how far your font size can be pushed before it interferes with your layout. You might need to adjust your style to accommodate these changes. How many levels you wish to account for is up to you, but a general rule of thumb is your page should support +3 with little disruption. You can of course offer special style sheets with links to enable them, giving you better control over the changes while still offering your users an adaptable experience.

Second is the ability to offer your users a text only version of the page. This can be done by applying custom style sheets for actions like printing or different devices that fit under our responsive design, which we will look at in more detail next.

Next is contrast. Color contrast between layered items like a link and a background color can be difficult for many users to distinguish and read. This is not to say that colors should never be layered, but that the contrast between them should be easy to distinguish. A helpful way to test this is to view your site with your brightness and contrast settings turned up and down a bit on your monitor to be sure your site is still legible.

Color combinations can also come into play, for example red and green, and green and blue. When these colors are used together, such as on a submit and cancel button that are next to each other, users with certain forms of color blindness can find them hard to read if not indistinguishable. A better approach is to use a cancel link of regular text next to a large submit button. This makes the options very visually distinct.

For example, the red/green form of colorblindness (Deuterope) makes red and green colors look more alike:



If you are counting on the color of the button to be an important indicator of function, that affect has been lost. Even worse, if you have combined text and background colors that do not lend themselves to readability for the colorblind, they may not be able to interpret the button's function at all!

These considerations are just some examples of a much larger topic, which is accessibility. The W3C maintains [a full list of items](#) to be considered to improve your site for the widest variety of users, and there are a number of sites that can scan your pages to provide notes and tips that follow the W3C guide, including one from the W3C itself at <http://validator.w3.org/>. Sites that pass the W3C validation can add “stamps” to their pages that identify their efforts and support, and to indicate that those considerations have been made.

This page titled [1.15: Font and Text Decoration](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [2.15: Font and Text Decoration](#) by [Michael Mendez](#) has no license indicated.

1.16: Responsive Styling

Designing for Different Screens

Our last rule concept of “When” can be applied to responsive design. It applies to rule conditions being, such as a user has a device with a mobile screen, or is trying to print our page. Since these are not within our default display, we may wish to remove elements or modify our layout to support a small screen, or remove background images and colors to make printed versions easier to read and less taxing on ink supplies.

To do this from within our style sheet, we need to specify the differences between our selectors that apply to these use cases. When we want to make style changes based on the user’s device, it is best to test for the existence of the feature in the browser, or for a specific feature or setting of a device, instead of the name and version that it identifies itself as. For example, if we consider the browser’s current width, it does not matter what the device is. A user could be on a desktop but have the browser much smaller than full screen. We also do not want to maintain a list in our code of every mobile phone size, especially when most of them fall within a limited range. By just checking the size, we can categorize the device and give it a best-guess-fit appearance.

Another argument for this approach is how the level of support for features changes both as standards change and as browsers are updated. If we only check the browser name, and a newer release than we were aware of supports the feature we want, we would lose out on the feature we want to implement. Instead, we can try and “test load” some of the features we want, or include fall back rules (like we did in [our gradient example](#)) to give ourselves the widest amount of support we can.

To test by feature or capability, we use the media queries feature of CSS. We will use screen size as an example. While not a perfect science, there are roughly 4 basic screens to consider, along with their average sizes:

- Smart phones (portrait)—320 pixels
- Tablets (portrait)—768 pixels
- Tablets (landscape), netbooks—1024 pixels
- Laptops, desktops—1024 or higher

This by no means captures every device or possible resolution. For example mini-tablets (or the trend of phablets/oversized cell phones) can fall around the 480 pixel range. Whether you want to support different layouts for these devices will determine if you need to consider more variety than the list above, but once we implement more responsive features you will see how these devices will usually receive an acceptable viewing experience. Next we will create new sections in our CSS that define what changes are necessary for those sizes. Some browsers and devices, like Apple’s Safari browser, will attempt to fit the entire page into their device screen by reporting to be a full size device. To prevent this, we can add the following meta tag to our HTML:

1. `<meta name="viewport" content="width=device-width; initial-scale=1.0">`

This line tells the browser to report its actual device width so we can provide the proper view.

Next, we need to create a series of selectors that will test for our different scenarios. Using media queries to guide our decision:

1. `@media only screen and (max-width: 959px) {}`
2. `// Smaller than desktop/laptop (tablets, netbooks)`
3. `@media only screen and (min-width: 768px) and (max-width: 959px) {} //Portrait or landscape tablets, netbooks`
4. `@media only screen and (max-width: 767px) {}`
5. `// devices smaller than portrait tablets (mobile)`
6. `@media only screen and (min-width: 480px) and (max-width: 767px) {} // mobile landscape to tablet portrait`
7. `@media only screen and (max-width: 479px) {}`
8. `// mobile up to landscape mode`

If you already have some CSS written, you can add these selectors below your existing code. Any rules in your existing code that are not overwritten by one of these selectors will remain in use, and become your “fall back” rules. This is why we did not include a media query for full size devices (although you could, if you wanted a different set of rules to be your fall back, say to assume a smaller device instead of a full size device as your default style).

Within each of these new selectors is where we would override our “normal” style to adjust our user experience. Anywhere we used floating divs will automatically adjust (within their parent container) as the window size changes. With these selectors, we can move other pieces of content around or even turn off some items that are unwieldy on smaller devices. Larger ads may be too large in screen size and file size for easy consumption on a mobile device. To disable it, we add a rule to the media section for smaller devices that hides that content. Imagine we have the following layout:

Logo		
Link 1 Link 2 Link 3	Flash ad Content	An ad Another ad
Contact information		

This layout could be represented by the following HTML and CSS:

```

1. <header></header>
2. <div id="container">
3. <div id="left">
4. <ul>
5. <li><a href="">Link 1</a></li>
6. <li><a href="">Link 2</a></li>
7. <li><a href="">Link 3</a></li>
8. </ul>
9. </div>
10. <div id="main">
11. <div id="splash">[flash with id "video"]</div>
12. <div id="content">Our text content here</div>
13. </div>
14. <div id="right">
15. <div class="ad">an ad here</div>
16. <div class="ad">another ad </div>
17. </div>
18. <br style="clear: left;" />
19. </div>
20. <footer>Contact Information</footer>

```

```

1. <style>
2. div{
3. border-style:dashed;
4. border-width:1px;
5. }
6. header{
7. background:url(ourlogo.png) no-repeat;
8. background-size:200px 100px;
9. background-position:center;
10. height:100px;
11. }
12. #left{
13. min-width:100px;
14. max-width:150px;
15. float:left;

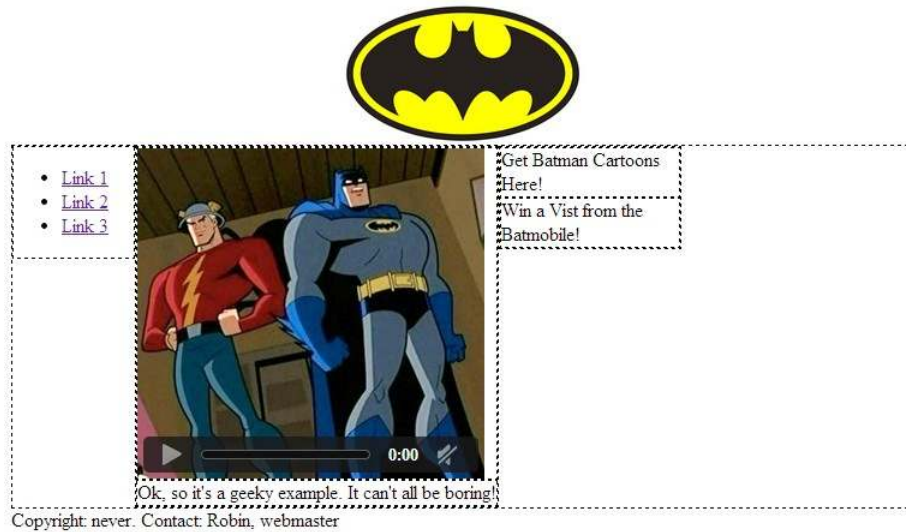
```

```

16. }
17. #main{
18. float:left;
19. }
20. #right{
21. min-width:100px;
22. max-width:150px;
23. float:left;
24. }
25. </style>

```

Which, in turn, could generate something like this once we add actual content:



Since we are using floated divs, the site will (eventually) conform to a top to bottom format that retains all content when it cannot fit across the display it is on. This is because float will place the elements in a line when the container is wide enough to hold them. When content is wider than the row, it is wrapped down to a new line. The max and min widths will help to ensure that content is not clipped, but while this helps us in some cases, the bulky flash video and large logo reduce the user's ability to browse from a mobile device. To adapt this example for users on smaller screens, we want to preserve all of the pertinent content while also preserving a good user experience. To do this, we will hide the video, put our links at the top, and move our ads to the bottom, all by adding media queries to our CSS. We will also opt for a smaller version of our logo. Since our current design was deemed flexible enough for other devices, we will just add the CSS selector for small screens:

```

1. <header></header>
2. <div id="container">
3. <div id="left">
4. <ul>
5. <li><a href="">Link 1</a></li>
6. <li><a href="">Link 2</a></li>
7. <li><a href="">Link 3</a></li>
8. </ul>
9. </div>
10. <div id="main">
11. <div id="splash">[flash width id "video"]</div>
12. <div id="content">Our text content here</div>
13. </div>
14. <div id="right">

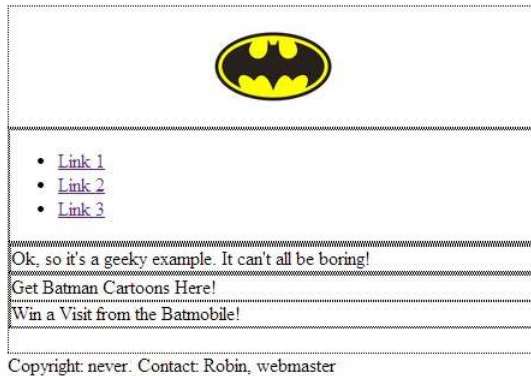
```

```
15. <div class="ad">an ad here</div>
16. <div class="ad">another ad </div>
17. </div>
18. <br style="clear: left;" />
19. </div>
20. <footer>Contact Information</footer>
21. <style>
22. div{
23. border-style:dashed;
24. border-width:1px;
25. }
26. header{
27. background:url(ourlogo.png) no-repeat;
28. background-size:200px 100px;
29. background-position:center;
30. height:100px;
31. }
32. #left{
33. min-width:100px;
34. max-width:150px;
35. float: left;
36. }
37. #main{
38. float:left;
39. }
40. #right{
41. min-width:100px;
42. max-width:150px;
43. float:left;
44. }
45. @media only screen and (max-width: 479px){
46. video{ display:none; }
47. #left{
48. width:100%;
49. float:none;
50. }
51. #main{
52. width:100%;
53. float:none;
54. }
55. #right{
56. width:100%;
57. float:none;
58. }
59. #header{
60. background:url(http://www.vectortemplates.com/raste...n-logo-big.gif) no-repeat;
61. background-size:100px 60px;
62. background-position:center;
63. height:100px;
64. }
```

65. }

66. </style>

Now, our floats and min/max rules will do the best they can until the device screen is less than 480 pixels. At that point our special style will kick in for mobile devices, putting our divs in a vertical row, hiding and resizing content, and making the divs as wide as the screen. Our site, without changing any HTML, will now look like the following:



We can also use media queries to set styling for when a user prints our page, without requiring them to click a link to a special version (selecting print from a browser menu would trigger this style) by using `@media print{ }`.

The full list of options for `@media` are as follows:

Table 1.16.1 Media Types

Name	Description
all	All devices.
aural	Text-to-speech readers.
braille	Tactile responsive devices.
embossed	Printing to braille printers.
handheld	Small portable devices.
print	Print view.
projection	Slides and presentations.
screen	Computer screens (regular size and up).
tty	Teletype terminals.
tv	TV displays.

1 <http://www.w3c.org>

2 http://www.w3schools.com/cssref/css_selectors.asp

3 <http://www.w3.org/standards/webdesign/accessibility>

This page titled **1.16: Responsive Styling** is shared under a **not declared** license and was authored, remixed, and/or curated by **Andrea Barrios**.

- **2.16: Responsive Styling** by **Michael Mendez** has no license indicated.

CHAPTER OVERVIEW

2: CSS and Styling a Web Page

Learning Objectives

1. The purpose of CSS
2. How to include CSS into an HTML file
3. What the page header and footer are used for
4. CSS syntax
5. The 4 types of CSS selectors
6. The difference between formatting and semantic tags
7. How to define a header section of a web page
8. How to combine CSS tags to reference specific sections of a document
9. How to include a separate CSS file
10. How to style a simple form

When creating a style for a web page, having a consistent look and feel to the application is important. The pages should look similar in format, and the behavior of buttons and menu options should work the same way across an application and suite of applications. Cascading Style Sheets (CSS) is the language used to style web pages, to make the pages look consistent and professional.

The form as it has been presented so far just contains the form elements, with no styling to make it look more presentable. This is in keeping with an age-old programmer prejudice that if something works, who cares what it looks like. But application development is about more than making the page look nice. It is about how to design a User Interface/User eXperience (UI/UX) that makes it easier and more pleasant for the user. A good thing for programmers to remember is a system that works but does not have users is as useless as a system that does not work.

Because of their implicit bias to simply make a system work, it usually does not fall to the programmer to do the UI/UX. But they must interface with UI/UX designers and need to respect what they do. That is why a basic understanding of CSS is important.

Like JavaScript, CSS is a language that is implemented inside of a web page. CSS uses the HTML `<style>...</style>` tags to contain the CSS attributes. The CSS attributes are mapped to HTML tags, ids, class names, or attributes to define how to style those elements.

This section will cover how to style the form developed in Chapter 2, and thus will present information about CSS as it is needed. At the end of the chapter some extra examples will be given to cover some useful ways to apply CSS for specific formatting.

[2.1: Web Page Header and Footer](#)

[2.2: Adding the form to the Page](#)

[2.3: Exercises](#)

This page titled [2: CSS and Styling a Web Page](#) is shared under a [CC BY](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

2.1: Web Page Header and Footer

When designing a user interface, an important part of creating a consistent experience is to design a consistent look and feel for the page. To create a look and feel, the first task is to develop and implement consistent page layout. Here that involves creating a header and footer section for the page.

To start the format for the header should be defined. A typical header presents information such as a title associated with the program or entity, other heading elements (such as the organization), a logo, and author information. Also, in the header is a menu of options for the system, which for now will contain the entries Home, File, and About. The header we will create for this page is as follows.

Figure 5 - Example HTML header



HTML provides two element selectors to make it easy to format the header and the footer of a web page, the `<header>` and `<footer>` tags. This section will format the header element.

4.1.1 CSS Syntax

CSS code is defined in a section of the HTML web page between `<style>` and `</style>` tags²⁵. Between these tags, CSS code is embedded into the HTML, just as JavaScript was embedded in the HTML. This CSS code contains display elements, or properties, that inform CSS how to render the page. The format for CSS is the following:

Program 70 - CSS syntax

```
<style>
  selector {
    attribute1 : value1;
    attribute2 : value2;
  }
</style>
```

In CSS, a selector is entered, and then associated with a block of properties to apply to that selector. There are 4 types of selectors that will be covered in this text²⁶. They are:

1. Element selectors, which are tags such as the `h1` and `p` tags.
2. Class selectors, which apply to a group of items. Class selectors start with a “.”, such as `header-icon` or `header-desc`, and are then referenced using the “`class=header-desc`” syntax in the element tag.
3. Id selectors, which apply to a single element referred to by its unique id. Id selectors start with a “#”, such as “`#inputForm`”. The id attribute in the tag is used to reference the selector.

4. Attribute selectors, which apply to attributes of a tag. For example, to make all textboxes which are `readonly` have a gray background, the following display attribute tag would be use:

Program 71 – Attribute Selectors

```
<style>
  input:read-only {
    background-color: lightgray;
  }
</style>
```

This attribute informs CSS to take any input tag which has a read-only attribute and make the background color gray.

Each of these selectors will be explained in context in the following sections that style the input form.

Display attributes, or properties, allow the programmer to describe how to display the information that will later be associated with the tag. Properties like font, background-color, border box, indentation, and literally scores of other attributes can be set. A complete list of all attributes that can be set is at <https://developer.mozilla.org/en-US/.../CSS/Reference>.

4.1.2 Semantic Tags

The `h1` is an element selector. It is also called a semantic tag.

Semantic tags are tags that are intended to simply provide formatting information, but have meaning about the information, or structure of the information, on the page web page. To better understand semantic tags, consider some of the tags in HTML that have been deprecated. For example, the `<b>` (bold) tag says what to bold the text (what to do with it). The `<strong>` tag describes the text as important, and to render the text as important, normally by bolding it. Similarly, the `<i>` (italicize) tag has been deprecated in favor of the `<em>` (emphasis) tag. In both of these examples, the new tags say something about the text, not something to do to the text.

The `h1` tag describes semantic information, that what is contained between the `<h1>` and `</h1>` is a specific part of the page (the header), and not some random division on the page. The header is the block at the top of a page, and the information associated with it is specifically how to format the header of the document.

4.1.3 Setting up the header block

The header of a document is the part of the document that is displayed at the top of the web page. The properties to define how the header should appear are set as display attributes when the header tag is defined. The header tag is defaulted as follows in CSS.

Program 72 – Header definition

```
<style>
  header {
    display: block;
  }
</style>
```

The default `h1` tag includes just one attribute, `display:block`, which specifies that the header sites in a block by itself at the top of the document. In CSS, a `block`²⁷ attribute means create a space that spans from the left to right margin of the element or page, allowing no other elements to be displayed to the left or the right. This is the normal behavior of a header, which generally spans the entire length of the top of the page. Like any display attribute can be overridden if needed.

The header for the example page developed in this chapter will be built in stages in the subsections below.

4.1.4 Changing the background and text colors

The first attributes to be change will make the box a dark color (we will use `slategray`) and make the text `white` . This is done by setting the attributes `background-color` to `slategray` , and `color` to `white` .

To make the header stand out more, a box will be placed around the header, using a 2-pixel large blue line. The header is also indented 50- pixels from the sides and top and bottom. This results in a first pass for the header in the following code.

Program 73 – Header attribute settings

```
<html>
  <head>
    <title>Please change this to the title of your page </title>
    <style>
      header {
        margin : 50px;
        border : 2px solid blue;
        background-color : slategray;
        color : white;
      }
    </style>
  </head>

  <body>

    <h1>This shows the header style.</h1>
  </header>
</body>
</html>
```

The page resulting from this program looks as follows:

Figure 6 - First pass at the web page header



4.1.5 Changing the font size using the

tag

The text inside of the header should be larger than the normal text in the document, to make it stand out. Normally text is placed inside of

(paragraph) tags in a document, and the styling attributes applied to the tag. The following example shows how the font-size can be changed for the tag to be 150% of its normal size.

Program 74 - Making the text 150% of the normal size

```
<html>
  <head>
    <title>Map Example Input Screen </title>
    <style>
      header {
        margin : 50px;
        border : 2px solid blue;
        background-color : slategray;
        color : white;
      }

      p{
        font-size : 150%;
      }
    </style>
  </head>

  <body>

    <h1>Map Example Input Screen</h1>

    Gettysburg Research Institute

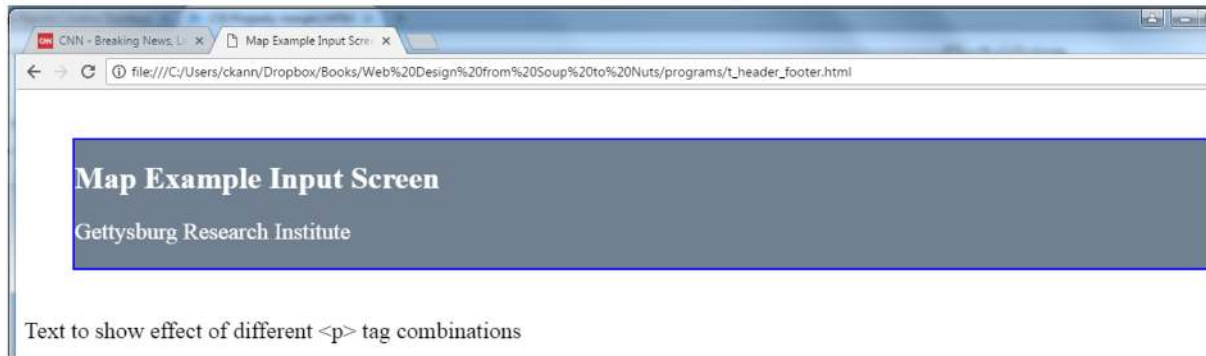
    </p>
  </header>

    Text to show effect of different
tag combinations
    </p>
  </body>
</html>
```

When running this page, it is obvious that all the text in the document is 150% of the normal size, not just the text in the header. Changing the

tag caused the text in all of the tags in the entire file to have the increased size of text

Figure 7 - Second pass at the web page header



To correct this so that the

tag will only affect the header, CSS allows tags to be combined so that changes to only have effect when use inside of a specific division or section of the page. The syntax for this is:

Program 75 - Combining the header and p tags

```
header p {  
    font-size : 150%;  
}
```

The code for the previous page has been changed so that only in the header does the tag change the size of the text to 150% of the normal size.

Program 76 - Paragraph text only affecting the header of the document

```
<html>  
  <head>  
    <title>Map Example Input Screen </title>  
    <style>  
      header {  
        margin : 50px;  
        border : 2px solid blue;  
        background-color : slategray;  
        color : white;  
      }  
  
      header p {  
        font-size : 150%;  
      }  
    </style>  
  </head>  
  
  <body>
```

Map Example Input Screen

Gettysburg Research Institute

</p>

</header>

Text to show effect of different

tag combinations

</p>

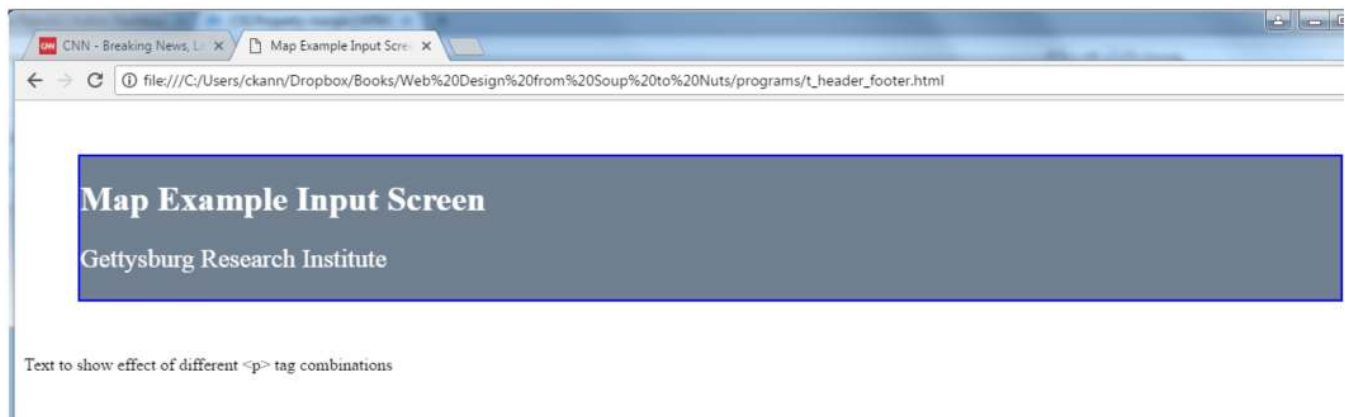
</body>

</html>

The result is the

tag only effecting the text in the header of the document.

Figure 8 - Third pass at the web page header



4.1.6 Dividing up the header block

In Figure 6, the header had 3 separate areas, one for the logo, one for the title and page information, and one area for the options to be implemented. To divide a web page into parts, a ²⁸ (or division) tag is used. The `div` tag is the most useful of all CSS tags, as it can be used to break up a web page into different areas, and to assign different styling or information types to those areas. For now, it will be used to break the header into 3 pieces.

To break the header up into 3 pieces, 3 divisions are created. These are all placed inside of the header section and can be thought of as *parts of* or *children of* the header section. This allows these 3 divisions are to appear inside of the web page header. These 3 division are shown below.

Program 77 - Dividing the header into 3 divisions.

```
<div >
  <image src="GRI_logo.png" />
</div>
<div >
```

```

<h1>Map Example</h1>

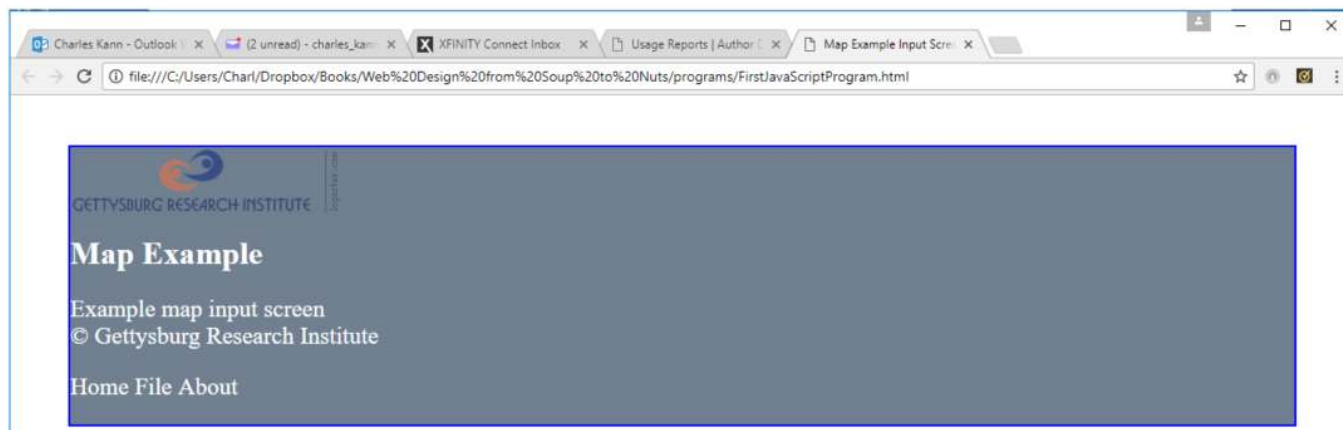
Example map input screen <br>
&copy; Gettysburg Research Institute
</p>
</div>
<div >

Home File About
</p>
</div>

```

This code only divides up the header so that the DOM knows they are 3 separate areas inside of the header. The DOM does not know how to display them, so it just places them in separate blocks, as shown below.

Figure 9
—
Un-styled div sections



To make the divisions work as intended, they must be styled.

To style each division, the divisions must be given a way to reference them. If there are a number of divisions to be styled the same, normally a `class` variable is defined in CSS, and referenced using the `class` attribute in the tags. If the division is to be uniquely styled, it will be given an `id` attribute, and CSS will style it using an `id` variable. Since the divisions in the header will be unique for the header, they will use `id` variables to reference and style them. Note in the program above, the individual divisions have been assigned the names `header-icon`, `header-desc`, and `header-menu`. This follows CSS conventions which favor hyphenated names.

When referencing `id` names in CSS, the name is prepended with a hash tag (`#`). For example, to set the `header-desc` to be a display of `inline-block` the following would be used.

```

#header-desc {
    display : inline-block;
    margin : 25px;
}

```

Careful readers will remember that this is how `id` variables were referenced in JQuery. JQuery uses CSS naming conventions to access DOM elements. This was alluded to earlier, but now is made explicit. Thus, to use JQuery at least a passing knowledge of

CSS is necessary to know how to access DOM elements

The main purposed of the CSS for these header divisions is to set the display parameters for them. The choice of inline-block tells the DOM to lay these divisions out as blocks, one after another on the same line. The margin tells the DOM how far to space the elements apart.

The last attribute set in the header divisions is the `float` attribute. The `float` attribute defines which side of the line to place the attribute on. Normally inline and inline-block elements start at the far left and are placed after each other from left to right. By saying `float: right`, the program is saying place this element starting at the far right, and then place subsequent elements moving from right to left. Menus items generally are place at the right of the header to balance the header.

The final CSS code for this example is shown below.

Program 78 - Completed header

```
<html>
  <head>
    <title>Map Example Input Screen </title>
    <style>
      header {
        margin : 50px;
        border : 2px solid blue;
        background-color : slategray;
        color : white;
      }

      header p {
        font-size : 150%;
      }

      #header-icon {
        display : inline-block;
        margin: 50px 10px 50px
      }

      #header-desc {
        display : inline-block;
        margin : 25px;
      }

      #header-menu {
        display : inline-block;
        float: right;
        margin : 75px 50px 50px 50px;
      }
    </style>
  </head>

  <body>
```

```

<div >
    <image src="GRI_logo.png" />
</div>
<div >
    <h1>Map Example</h1>

    Example map input screen <br>
    © Gettysburg Research Institute
</p>
</div>
<div >

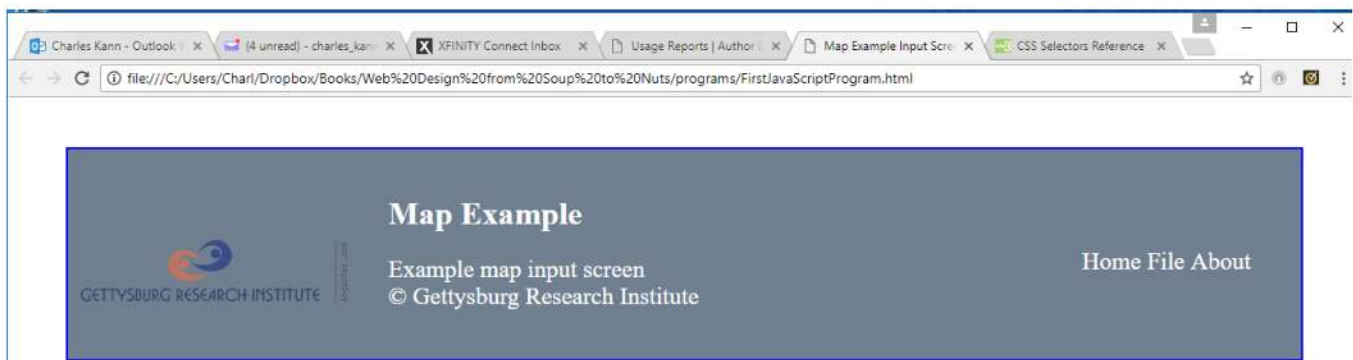
    Home File About
</p>
</div>
</header>

</body>
</html>

```

This program gives the final, completed header for the page.

Figure 10
—
Completed web page header



4.1.7 Managing CSS

To make managing a project and writing and reading source code easier, the CSS source code for a web page is generally kept in a separate file from the html source. This file separation achieves a number of positive benefits: 1) it keeps the styling information separate from the application information, which makes it easier to read and understand the HTML program; 2) By separating the styling and program, UI designers can work on the styling the application without interfering with the programmers developing the application.

For the header developed above, the file containing the CSS could be kept in the file WebMapExample.css, and the HTML could be kept in the file MapExample.html. The CSS file is included in the web page using a `<link>` tag.

```
<link rel="stylesheet" type="text/css" href="WebMapExample.css">
```

The contents of the WebMapExample.css file is:

Program 79 - WebMapExample.css file

```
header {
    margin : 50px;
    border : 2px solid blue;
    background-color : slategray;
    color : white;
}

header p {
    font-size : 150%;
}

#header-icon {
    display : inline-block;
    margin: 50px 10px 50px
}

#header-desc {
    display : inline-block;
    margin : 25px;
}

#header-menu {
    display : inline-block;
    float: right;
    margin : 75px 50px 50px 50px;
}
```

The application file, MapExample.html, is now much simpler, and easier understand.

Program 80 - MapExample.html

```
<html>
  <head>
    <title>Map Example Input Screen </title>
    <link rel="stylesheet" type="text/css" href="WebMapExample.css">
  </head>

  <body>

    <div >
      <image src="GRI_logo.png" />
```

```
</div>

<div >
    <h1>Map Example</h1>

    Example map input screen <br>
    &copy; Gettysburg Research Institute
</p>
</div>

<div >

    Home File About
</p>
</div>
</header>

</body>
</html>
```

In the real world, most programmers will deal with the HTML and JavaScript, and UI/UX designers will handle the CSS for a web page. However, both sides, programmers and UI/UX designers, should know enough about the other technologies to be able to interface effectively with their counter parts.

4.1.8 Quick Review

1. What are the four types of CSS tags? Give an example of each and describe how you might use them.
2. In your own words, describe *semantics*? What is an HTML semantic tag, and how does it differ from a formatting tag?
3. What is the difference between a `display:block` and a `display:inline-block` attribute? How were they used to set up the header division of the web page?
4. How would you hide a division of an HTML document?
5. Looking at the following web page, https://www.w3schools.com/cssref/pr_class_display.asp, discuss how you might use the `display` attribute to format a web page?
6. How might you specify a paragraph (`<p>`) tag so that it only impacts the text in the `#header-desc`?
7. How would you format the `<div>` tag so that all of the output in the document associated with an error message would be in red?
8. Create a CSS file and include it in another HTML file. Does the name have to be “.css”?

²⁵ CSS styling can also be done directly in an HTML tag using the *style* attribute, but this violates the idea of separating concerns in a web page, and like Unobtrusive JavaScript, best practice says to not mix CSS and HTML.

²⁶ For a complete list of all selectors and how to use them, see https://www.w3schools.com/cssref/css_selectors.asp.

²⁷ For a complete listing of all display types, see https://www.w3schools.com/cssref/pr_class_display.asp.

²⁸ The current thinking about CSS is that an HTML document should be divided into divisions (using the `div` tag) when the page is being divided for styling purposes, and into sections (using the semantic tag) when it is being divided into parts that would be equivalent to areas in an outline. There is no difference other than the semantic meaning of the section as part of a larger document

organization. As this document is concerned with formatting program web pages, it will use only divisions to keep the confusion to a minimum.

This page titled [2.1: Web Page Header and Footer](#) is shared under a [CC BY](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

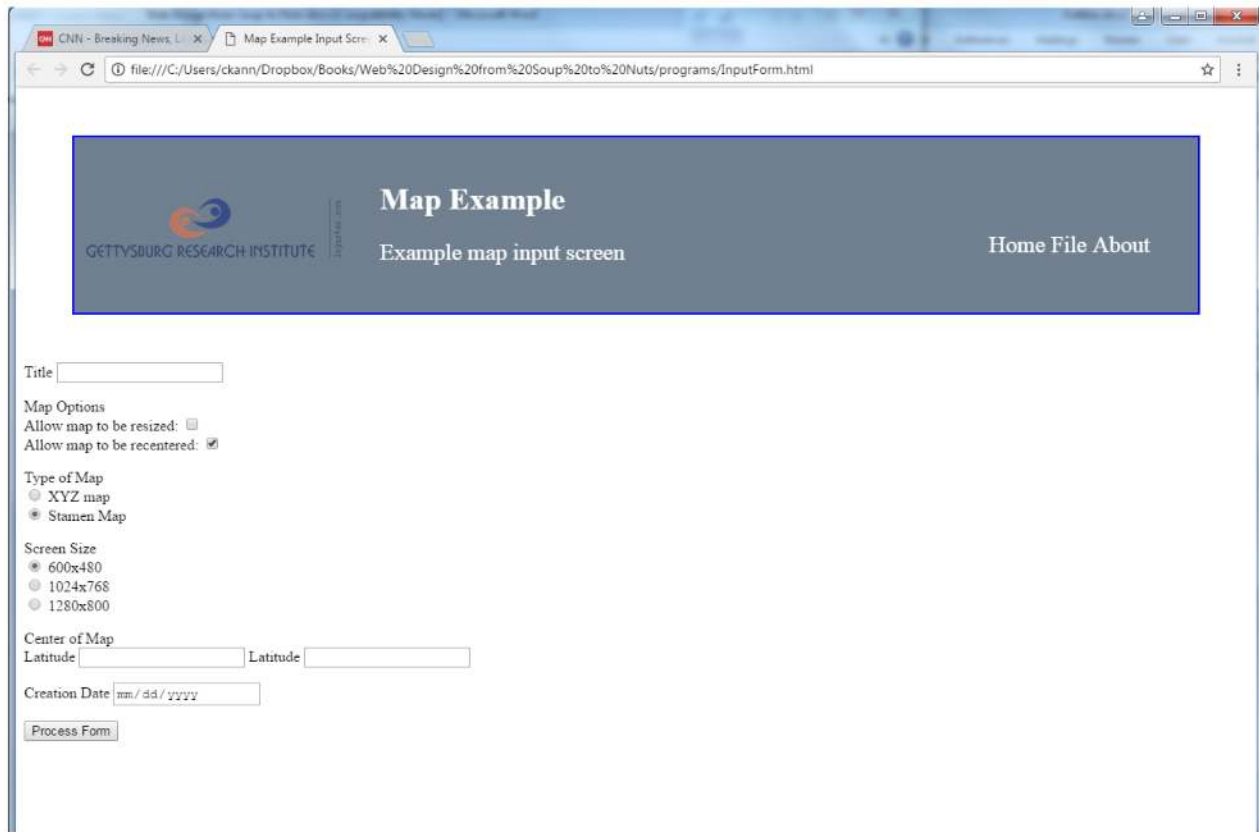
- [4.1: Web Page Header and Footer](#) by [Charles W. Kann](#) is licensed [CC BY 4.0](#).

2.2: Adding the form to the Page

After the header has been developed, the form is added to the page. For now, only the display of the form will be presented to make the concepts clearer. Later, this chapter will show how CSS can be used with JavaScript to control display of the form.

If the form is simply added to the page, as shown below, it looks very unprofessional. While it is functional, it is ugly and would not inspire confidence from the users that the web page actually works.

Figure 11
—
Completed web page header



The screenshot shows a web browser window with two tabs: 'CNN - Breaking News' and 'Map Example Input Screen'. The address bar shows the file path: 'file:///C:/Users/ckann/Dropbox/Books/Web%20Design%20from%20Soup%20to%20Nuts/programs/InputForm.html'. The page content is as follows:



Map Example

Example map input screen

[Home](#) [File](#) [About](#)

Title

Map Options
 Allow map to be resized: ☐
 Allow map to be recentered: ☒

Type of Map
☐ XYZ map
☒ Stamen Map

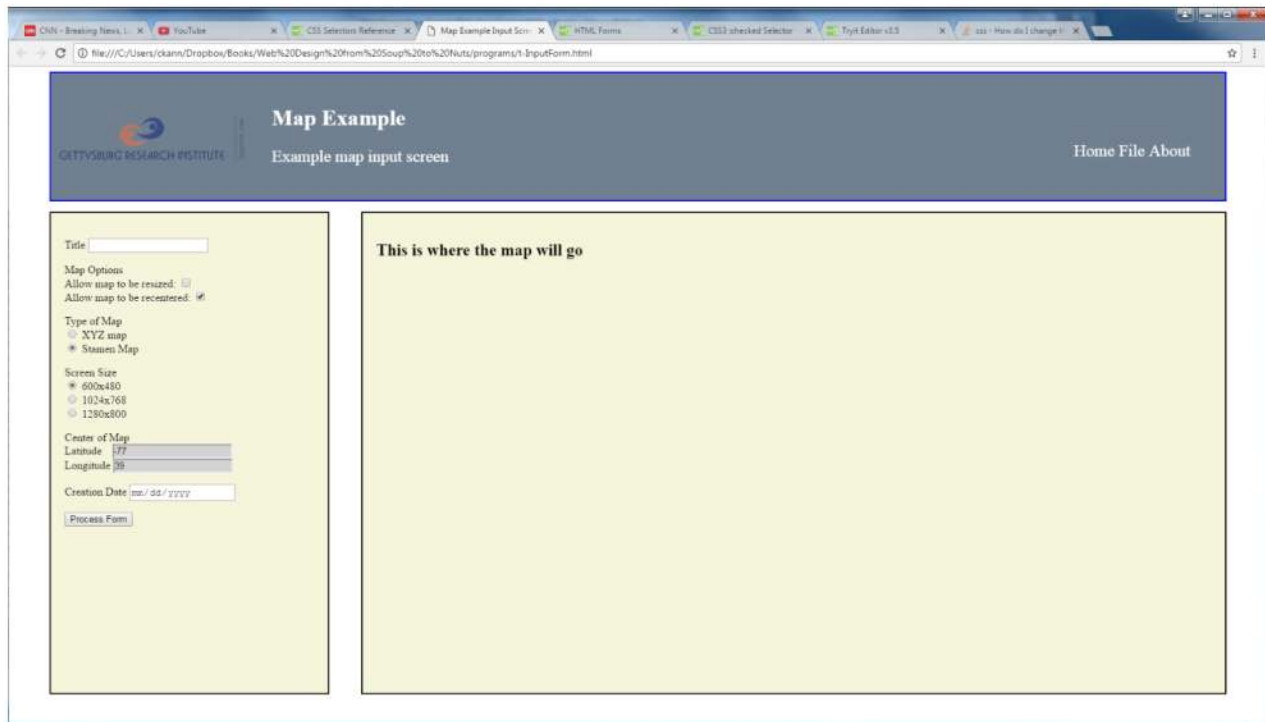
Screen Size
☒ 600x480
☐ 1024x768
☐ 1280x800

Center of Map
 Latitude Longitude

Creation Date

A much nicer layout of the form follows. The following is the form display that will be styled using CSS. Following the form, the CSS and HTML are presented here and an explanation of all the elements is given.

Figure 12
—
Completed web page header



CSS

Program 81 – Form CSS example

```
header {
    margin : 5px 50px 5px 50px;
    border : 2px solid blue;
    background-color : slategray;
    color : white;
}

header p {
    font-size : 150%;
}

.header-icon {
    display : inline-block;
    margin: 50px 10px 50px
}

.header-desc {
    display : inline-block;
    margin : 25px;
```

```
}

.header-menu {
    display : inline-block;
    float: right;
    margin : 75px 50px 50px 50px;
}

#inputForm {
    margin : 10px 5px 5px 50px;
    background-color : beige;
    border : 2px solid black;
    display: inline-block;
    float : left;
    padding : 20px;
    width : 20%;
    height : 70%;
}

#map {
    margin : 0px 50px 5px 5px;
    background-color : beige;
    border : 2px solid black;
    display: inline-block;
    float : right;
    padding : 20px;
    width : 67%;
    height : 70%;
}

input:-moz-read-only { /* For Firefox */
    background-color: lightgray;
}

input:read-only {
    background-color: lightgray;
}
```

The CSS for the head has been defined, and that leaves just 4 CSS tags to cover here. The first are the `#form` and `#map` id tags. These two tags create boxes displayed on the web page to contain other form elements. The `#form` is 20% of the size of the page, and the `#map` is 67% the size of the page and is anchored to the right of the page using the `float:right` attribute. The `inline:block` attribute tells CSS to put both on the same line. Since they only take up 87% of the line, 13% of the line will not be included in either division and left blank between the two pages. Both or the div sections have black boarder 2 pixels in size, and both will take up 70% of the height of the page, with the rest used by the header. Finally, the padding says leave some room on the right and left of the divisions before rendering the text.

The final two tags, the `input:-moz-read-only` and `input:read-only` tags, are to gray out any input tags which are read only, e.g. the value for these input fields cannot be typed in by the user and must be set programmatically. These are the read only fields on the map for latitude and longitude.

Program 82 - CSS with form completed example.

```
<html>
  <head>
    <title>Map Example Input Screen </title>
    <link rel="stylesheet" type="text/css" href="WebMapExample.css">
  </head>

  <body>

    <div class="header-icon">
      
    </div>
    <div class="header-desc">
      <h1>Map Example</h1>
      <p class="header-p">
        Example map input screen
      </p>
    </div>
    <div class="header-menu">
      <p class="header-p">
        Home File About
      </p>
    </div>
  </header>

  <section />

  <label for="title">Title</label>
  <input type="text" size="20">
</p>

  Map Options<br>
  <label for="resize">Allow map to be resized:
</label>
  <input type="checkbox" />
  <br/>
  <label for="recenter">
    Allow map to be recentered:
  </label>
  <input type="checkbox" checked />
</p>
```

```
Type of Map<br>
<input type="radio" name="maptype" value="XYZ Map"/>
<label for="XYZMap">XYZ map </label>
<br/>
<input type="radio" name="maptype" checked />
<label for="StamenMap">Stamen Map
</label>
</p>

Screen Size<br>
<input type="radio" name="screenSize" checked
  value="600x480" />
<label for="XYZMap">600x480 </label>
<br/>
<input type="radio" name="screenSize" value="1024x768"/>
<label for="XYZMap">1024x768 </label>
<br/>
<input type="radio" name="screenSize" value="1280x800"/>
<label for="XYZMap">1280x800 </label>
</p>

Center of Map<br>
<label for="lat">Latitude
</label>
<input type="number" value="-77" readonly /><br />
<label for="long" >Longitude </label>
<input type="number" value="39" readonly />
</p>

<label for="creationDate">Creation Date
</label>
<input type="date" />
</p>

<input type="button" value="Process Form" />
</section>

<section >
  <h1> This is where the map will go </h1>
</section>
</body>
</html>
```

Figure 13
—
Completed web page header



The screenshot shows a web browser window with multiple tabs. The active tab displays a web page titled "Map Example" with the subtitle "Example map input screen". The page header is dark blue and contains the "GETTYSBURG RESEARCH INSTITUTE" logo, the title "Map Example", the subtitle "Example map input screen", and navigation links "Home File About". The main content area is divided into two sections: a left sidebar with form controls and a large right area for the map.

Form Controls (Left Sidebar):

- Title:
- Map Options:
 - Allow map to be resized: ☐
 - Allow map to be recentered: ☒
- Type of Map:
 - ☐ XYZ map
 - ☒ Static Map
- Screen Size:
 - ☒ 600x400
 - ☐ 1024x768
 - ☐ 1280x800
- Center of Map:
 - Latitude:
 - Longitude:
- Creation Date:
-

Map Area (Right):

This is where the map will go

For the HTML for the page, the only change was to add two section tags to create sections for the form and the map. The section tag is a div tag that has a semantic meaning. A *section* to a form is a completely separate part of the form. A *division* has no semantic meaning.

This page titled [2.2: Adding the form to the Page](#) is shared under a [CC BY](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [4.2: Adding the form to the Page](#) by [Charles W. Kann](#) is licensed [CC BY 4.0](#).

2.3: Exercises

Take the form you developed in Chapter 3 and style it using CSS.

This page titled [2.3: Exercises](#) is shared under a [CC BY](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [4.3: Exercises](#) by [Charles W. Kann](#) is licensed [CC BY 4.0](#).

CHAPTER OVERVIEW

3: Trying It Together

Learning Goals:

By the end of this section, you should be able to demonstrate:

- The ability to create a site that incorporates multiple languages
- The ability to extract information from a database and present it in a site
- Basic security considerations to deter malicious attacks
- The ability to interact with user data and alter a page as a result

[3.1: Security](#)

[3.2: Integration Examples](#)

[3.3: Finishing Touches](#)

[3.4: Now What?](#)

This page titled [3: Trying It Together](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

3.1: Security

Risk

It can be easy to both over and under consider the security of your website. New initiates tend to do everything possible to secure their code, which often results in overblown measures that can take considerable time to incorporate. This may only serve to protect something that may not require security in the first place. The dangers, then, are: getting tired of the complicated process, becoming complacent because nothing has happened, or neglecting to consider security in the first place. As a result, systems end up exposed.

Risk management is the practice of finding the sweet spot where the time and cost of implementing the measure is proportionate and acceptable to the perceived threat a compromise would create. Risk analysis is another of the many topics in this text that begs for much deeper study, and I would encourage everyone to read beyond what is here as it applies to everyone involved in a programming project.

In short, some things to consider about your system are its level of exposure, cost to acquire/replace its contents, importance of contents in relation to the company, and importance of the contents in relation to your clients. A number of industries are also bound to compliance measures based on certain types of information, or in order to achieve certain certifications.

Some questions to ask are things like:

1. Is this system only available on our network, or is it publically exposed?
2. Is this system only accessible through other security mechanisms like credentialing systems, SSL, etc.?
3. How much would it cost to replace the data the system holds?
4. Would it be possible to replace the data if it is lost?
5. Does the value of the data change if it is stolen or made publically available?
6. If the data is compromised, how would it affect our relationship with our customers?
7. Is any of the information required to be secured to a certain level?
8. Are we an industry (e.g., healthcare) that has to comply with a federal or other reporting guideline (e.g., HIPAA)

Answering these questions can be difficult, as you need to find quantifiable measures to questions that are better suited to qualitative analysis. An example of how to approach this is to determine risk values that address each topic. If data is irreplaceable, it might be an 8/10. If compromise poses no risk if stolen or exposed, it may be a 1/10. Using the same min/max range for each value allows us to add up our scores in order to get a summary value of the risk. This value can help you see the importance of your system better. From here, you can determine based on its value, particular values, or “red line” items that require specific minimums like federal requirements, exactly what security implementations you will need. You can also create estimates of what damages (data recovery costs, man hours to fix/replace the system, loss from lawsuits, etc.) might cost against that of implementing specific security measures as a cost/benefit ratio. For example, a mitigating effort of installing or developing a credentialing system might cost \$5,000 dollars, but if the data is exposed a lawsuit might entitle the plaintiff to \$80,000 dollars in damages. How much is spent to defend a system in relation to what the perceived losses might be will ultimately be determined by your executives and what the analysis reveals.

A common response to this issue is what if the system never gets attacked? Then the costs would be wasted! Alas, in many cases it can be hard to know or prove that risk management efforts actually stopped or prevented an attack. Time and money spent on security are difficult to defend, as their return on investment can rarely be proven. Was the system not compromised because of your efforts, or because no one tried? The typical reaction is to consider the survivability of the organization if the system is compromised or lost, and this is where strong metrics and analysis can save the day, the system, and maybe even your job.

Our example above of measuring each item on a fixed scale to determine a single threat value is just one mechanism. It is good for quick (relatively speaking) analysis and as a means of measuring multiple systems, or different possible versions of a system, against each other. More advanced techniques take more factors into consideration. For example, [the NIST SP 800-301](https://nist.gov/publications/detail/nist-sp/800-301) analysis includes a threat-likelihood-impact matrix where values are assigned to each element (the amount of likeliness of a given threat compared to its impact), repeated for each threat. Placing things into the matrix helps to quantify the scenario.

To perform a full risk analysis, you will want to consider all possible sources of damage to your system. These include intentionally malicious or accidental actions of users both in and outside of your company (just because a user is authorized to access data does

not mean it is being used correctly or as intended), natural or man-made disasters that would affect your system (floods = water; water + electronics = bad), or just general failure (power surge + electronics = bad). For the remainder here, we will only focus on the former as it directly applies to the topics we address in the rest of the text.

To delve deeper, look at some of the United States federal risk management initiatives.

- [National Institute of Standards and Technology \(NIST\)](#)²
- [Department of Homeland Security \(DHS\)](#)³
- [The United States Computer Emergency Readiness Team \(US-CERT\)](#)⁴

One parting note: No matter the size of your organization, even if you are the lone programmer, a risk analysis exercise should involve many minds, especially those of upper management. Their determinations will weigh heavily on your activities, and their exposure to these exercises will increase their awareness overall.

Now that we have an idea of how to discover and quantify risks, we can look at some basic methods of protecting ourselves.

PHP

One of the easiest ways for a malicious user to attempt to take advantage of our database, or gain unauthorized access, is to use our own scripting language against us. Let us consider a basic example. Say you have a form that takes a username and password to log in a user. Even if you employ an encrypted password with hashing, a malicious person could still take advantage of your form. Since your form action and field names are available to anyone who views your page's source code, they could read your form to generate the following URL: `http://yoursite.com/youractionpage.p...password=12345'` or `1`

Since they are using your form's variable names, your action page will, in this basic example, assume they are legitimate. The key element to note here is the apostrophe or `1` portion of the password field. If you are accepting user input without checking it first, like this:

```
1. mysqli_query("select * from users where username='$_GET[username]' and password='".md5($_GET[password])."'");
```

Your populated query would actually read as follows:

```
1. select * from users where username='FAKEUSER' and password='12345' or 1
```

When evaluated, the malicious user is automatically logged in because the “or 1” makes the logic statement true, since “or true” will always be true, regardless of the first half of the statement. Your malicious user is now logged into your site! If you did not follow the “least needed privileges” theory (we will look at this next), they would then have access to anything in the database that the account is allowed to access—potentially even other databases.

To protect against this, we can use [the sanitization](#)⁵ and [validation](#)⁶ features of PHP to ensure our users' responses are valid.

For example, if we ask a user to enter their email, we can first remove anything that should not be in an email address:

```
1. $email = filter_var($_GET['email'], FILTER_SANITIZE_EMAIL);
```

And then make sure it is still in proper format:

```
1. if(filter_var($email, FILTER_VALIDATE_EMAIL)){echo "OK!";}
2. else{ echo "Please re-enter email!"; }
```

The above code would be OK for a regular email address, but anything missing an `@`, or a top level domain like `.com`, would ask for it to be re-entered. Our sanitization for an email address will remove all characters except letters, digits and `!#$%&'*+./=?^_`{|}~@.[]`. You typically want to sanitize before validating. If your sanitization removes something that was actually a desired part of the input, the validation would then fail.

MySQL

The source (for us, at least) of the risk: the data itself. We are storing our data in MySQL, which our web pages interface with in order to build our interactive experience. When we store user credentials in our database, we need to consider the fact that since our website communicates with the database, there is an inherent weakness. This is true whether or not our database is on the same physical system, or lives in the same operating system, as the site itself. One of the basic mechanisms to protecting your site is to

prevent unauthorized access. Part of this can be achieved by obfuscating our users' passwords when we store this. To do this, we can employ a couple of tactics called hashing and salting.

Hashing is the process of passing the password through a mathematical algorithm that turns the password into a much longer string of characters. The algorithm is designed to be one-way, meaning a hash cannot be passed to an algorithm that reveals the original text. The algorithm will produce the same hash every time for a given password (although occasionally duplicates may occur, a fault called a "hash collision"). The result is stored in the database in place of the user's actual password. Each time the user logs in, they type in their original password, your authentication system passes it through the algorithm, and you can compare the result to the stored value to make sure they match.

The use of a hash ensures that if someone is able to gain access to your database, they cannot simply copy the credentials and use them to their advantage, as the stored value is not the actual password they would use. A couple of popular methods that have been used to achieve this are MD5 and SHA-1, among others. I discourage the use of MD5, as its hashes are comparatively shorter than many others in use today, and enough data has been gathered about the algorithm that many password hashes can already be found just by using sites like md5decrypter.com.

Another issue of only relying on the original hashing algorithm is that other sites may be doing the same. If someothersite.com and your site are both using MD5 by itself, and someothersite.com is compromised, the attacker would have the usernames and passwords for their site. If any of your users also used someothersite.com, odds are some of them are probably using the same email address, username, and password. With all of this data, the attacker now has a set of usernames and passwords to try on your site. Since you are using the same hashing mechanism, the credentials used on both sites would work on yours as well.

The hashed value of the word password, for example, returns to us the value 5f4dcc3b5aa765d61d8327deb882cf99. If you take this value and paste it into md5decrypter.com, you will see our original text (password) given back to us.

Never fear, however, as we can defend against this with a salt. Salting is the process of adding something extra to your user's password, like adding salt to food to make it better. This should ensure that even if a user has the same password on the other site, the one stored in our database is still different. This means a malicious person cannot use the encrypted password to get into our site.

If the salt for our user was #hsy5, our user's password that we would hash could be password#hsy5 or #hsy5password or even pass#hsy5word. When we hash password#hsy5, we get 5b48480a7171f41d2bf52093f4850281. Now, if someone tries to use known passwords, their hash for password will not work on our site. You can use salts by either creating a salt for every password, or using one that is coded directly into your script.

Keep in mind that if you are using a single salt for all values, make sure it is not running in a script the user would have access to, like a linked JavaScript value, or would appear in your HTML as a hidden value in a form. This gives them the missing key to circumvent the extra protection you added. By only having the salt in one script, malicious agents would need to gain access to your web server as well as your database, especially if this script is segregated from your database's home system. Storing a unique salt for each password means if one salt is discovered, not every account has been compromised. If the compromised account allows reading the salt table though, other accounts would eventually be accessible. Ultimately, using the best hashing algorithm available and both salting approaches would provide a high level of protection. How much you actually need to implement should be determined by completing a risk analysis.

Another strong self-protection method is to make sure your user accounts in your database are as locked down as possible. By this I mean abiding by a "least-needed security" approach. It is easy when developing to create a user account for your project with a simple password and wide open permissions to make development easier. Once you move your project live, however, you should take away all the permissions your users would not need and should not have. For example, your basic end users will likely not need the ability to alter or drop tables, so you should remove user access to those features. This means that if injection methods are used on your site, even if they succeed, the malicious threat still could not drop or alter your tables without also finding a vulnerability in your database.

To provide administrative users with an interface, you can set those pages to utilize a database user with higher credentials. The most sensitive operations (full database deletions, creating new administrators, etc.) should still be left in the hands of users who log into your database directly from a secured machine.

In MySQL, the full list of permissions elements are: *select, insert, update, delete, index, alter, create, drop*

Administrative privileges are: *create temp table, file, lock table, process, reload, replication, show dbs, shutdown super.*

We can quickly assign users the initial set of actions or the administrative set by using the keywords “usage” or “all” when assigning the account. Most sites can achieve everything necessary with the limited permissions of select, insert, update, and maybe delete. Keep in mind although overall this is quite restrictive, without proper precautions injection attempts could still use insert or update to infiltrate your site.

Depending on the element, security can be restricted to the user based on the levels of global (your entire database server, i.e., every database), database, table, or column. This granularity allows you to ensure users can only change what they need to.

Our database *de jour* for this text also allows us to require SSL connections, providing greater security between the user (an admin or our webserver) and the database. MySQL also allows us to limit the number of updates over a span of time, as well as the number of concurrent connections and queries per hour. Typically, we would want these values very high to support the highest volume of end users possible, but you may have a use case where you know these numbers should be in a certain range. For example, if your company has 50 employees, and only 40 need your site, then you could cap your concurrent users to 40. Anything beyond that could be an indicator of database problems or hacking.

Types of Hashes

There are a number of strong hashing algorithms available. Some of these are considered out-dated and are easily to reverse, like DES. Others are still widely in use even though databases of known encrypted/unencrypted values for many passwords are freely available, like md5. Sites like md5decrypter.com are examples of this weakness. For this reason, you will likely want to favor the newer approaches available to you at the time. As new encryption algorithms are created, the encrypted string lengths become much larger, making the computation power required to reverse them too time consuming (if even possible) for the current hardware available.

Credentials

Never say never, but never give root. Root, or the default user on your MySQL server, has the highest level of permissions in the system. This means a PHP script that uses the root account to establish a connection can do anything, even drop all databases. Because of the damage an accidental or malicious command can have on your data, any MySQL user account that your web server utilizes to interact with your data should be as limited as possible. The vast majority of applications can get by with simple CRUD actions (that is create, read update, and delete). In most implementations, I would go one step further and not even allow deleting, replacing it instead with an “inactive” flag for records to hide them from the user as if they were deleted. Depending on the system, your web users might need additional features like temporary tables, and these should be allowed only as needed. This means if the attacker tries to perform administrative type actions, and you are not preventing them, they will still be blocked as the user they are acting as will not allow it.

JavaScript

We always (Yes, always—laziness breeds poor security in our world) want to take a look at anything a user gives us before we interact with it. This is for two reasons: first, the user may have made a mistake. Maybe they mistyped an email address, or left a required field blank. Or, perhaps, the user is a malicious person or script attempting to do something other than what we intend with access to our site. It might be using our forms to spam others, gain access to our data, or make unsolicited changes to our site. We already discussed this under PHP but we can attack the problem with JavaScript too.

Taking this into account, before we use anything a user gave us, we need to make sure (as much as possible) that it is safe data to interact with. Usually we want to do as much of this as possible on the user (or client) side, so they do not need to click submit and wait for a response from the server to find out something is not quite right. To do this, we can use client-side scripting like JavaScript to make sure things are OK as we progress. As fields are changed, JavaScript can look at the content and make sure addresses are formatted correctly, required fields are filled in, etc. Coloring, highlighting, or providing messages to the user when problems occur. We can achieve this easily by tapping into jQuery’s validation library:

1. `<script src="/lib/js/jquery.validate.js"></script>`
2. `<script>`

```
3. $(document).ready(function){$("#commentForm").validate(  
4. cname : { required : true, minlength: 2 }  
5. );}  
6. );  
7. </script>
```

This example would execute the validation once the form is loaded, showing that the cname field is required and the minimum length is two characters. Not only can jQuery help us display these requirements on the form itself, we can call the validator as fields are changed and/or when the form is submitted before leaving the page to enforce the rules we provided.

In terms of user experience, this is typically done in real time. As soon as a user leaves a field, the script makes sure it is OK, and provides confirmation of the fact (typically a green highlighting or “OK!” type of marker) or by not marking the field as bad (typically red, or prompting the user to re-enter the field).

Once the form is completed, JavaScript should ensure that the user’s submission will be good on the first try (at least content-wise—we cannot confirm things like a username and password without talking to the server). This accounts for our number one concern: mistakes from the user. Even though we checked the submission, we want to repeat this process on the server-side in more depth. If the user is malicious they may be circumventing our page, or the user may have JavaScript disabled.

The server-side script should take into account the nefarious user. If someone tried to subvert our form, JavaScript probably caught it. If, however, we are using GET or they use a script to send data directly to our action page from our form (which they can easily find in our page source) then they can get around our JavaScript.

Execution Functions

Both PHP and JavaScript support features that allow the user to access and run other programs or scripts on the web server or local system. This can be useful when you want to interface with another application or system that the language does not have the ability to communicate with directly, but it exposes a huge security risk. Anything passed to these functions will be executed as if that user was sitting at the command prompt of your web server. The implications here are fairly obvious, as anything your server’s “web user” account has permission for would be allowed. If you are passing a variable into the execute function, you have created a path directly to the heart of your system.

The best bet is to avoid using these entirely unless absolutely necessary. If you must, ensure that variables are not passed to the function if at all possible to prevent injection. Finally, if all else fails, sanitize and validate anything passed, limit your web server user role as much as possible, and keep your system as up to date as possible to deter hackers.

In PHP you will want to avoid the `exec()` function. JavaScript is a bit more removed, but some actions can create the ability, such as creating an ActiveX object:

```
1. <script>  
2. var wsh = new ActiveXObject("WScript.Shell");  
3. wsh.run('notepad.exe');  
4. </script>
```

Segregated Systems

Your database server, ideally, would not live under the same operating system as your web server. This does not mean the same OS cannot be used on both systems, but that they are not residing on the same exact installation. This is important because if your system is exposed to, or faces, the Internet it is at a higher risk of compromise. Keeping the database within your network with a single controlled access point between the two means your data is not as compromised if the web server is.

Learn more

Keywords, search terms: Web server security, risk management, secure programming

76 Tips for Securing Your Server: <http://www.rackaid.com/resources/server-security-tips/>

Apache’s Security Tips: http://httpd.apache.org/docs/2.2/misc/security_tips.html

Symantec's Tips for MySQL: <http://www.symantec.com/connect/articles/securing-mysql-step-step>

This page titled [3.1: Security](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [5.1: Security](#) by [Michael Mendez](#) has no license indicated.

3.2: Integration Examples

The following code examples will demonstrate how the languages we have studied can be combined to create dynamic systems. In each of these examples two or more elements covered in the text will be combined. These examples are not intended to function fully based on the excerpts you will see, but are meant to demonstrate methods of integrating the languages.

Connecting to MySQL

In order to have our users interact with our database, we need to establish a bridge of communication between them. To do this, we will create a connection to our database and store it in a variable. Through PHP we have a variety of methods of creating this connection, among them are libraries called MySQL, MySQLi, and PDO. These libraries have different approaches to interacting with a database, and there are differences in what commands are available and how the connection is treated. While you will see many examples online that use the standard MySQL connector, I would warn you away from it. It is on its way to deprecation, and has little injection protection built in. The MySQLi library is the one we will focus on in our examples here as it is a better start for entry level programmers. Ultimately, once you are comfortable with integrating these languages, I would recommend moving to PDO. While it is not tailored for MySQL, it supports a wider range of SQL databases that will allow you to more easily change your backend system.

To begin, we will call a function to create our connection. The shortest avenue to do this is as follows:

```
1. $mysql = mysqli->connect("localhost","user","password","database");
```

By inserting your server's values in each set of quotes, the variable `$mysql` will become our line of communication to our MySQL database. When we created our connection by using a class method, our `$mysql` variable is now a MySQLi object. We could also have used procedural style with `mysqli_connect`. Assuming your database is the same system as your website, "localhost" or "127.0.0.1" should work just fine for you. Your username and password can be any account that exists in your SQL database. In a fresh installation, "root" as the user and "root," "password," or nothing—" " as a password will usually let you in, but as we saw in security, you should avoid this unless necessary and only on a low-risk machine. The declaration of the database we want to use is optional, but saves us from having to select one later or always declare our database in our queries.

In the spirit of this section, we will revise this example to make it more useful. By replacing our values with variables, we can keep our actual values apart from the rest of the code. Additionally, we can separate our connection out to its own file. By doing this, we can `require()` or `include()` it on any page that we need a connection. Then all we need to do when we use our database is remember to close the connection at the bottom of our page. An additional advantage is that we could also wrap our connection in a class of our own, allowing us to re-declare from our own class what functions are available. For now, we will keep it simpler:

```
1. $host = "localhost";
2. $user = "username";
3. $password = "password";
4. $dbase = "database";
5. $mysql = mysqli->connect($host, $user, $password, $dbase);
```

If this code is its own file like `database.php`, we could use it in all of our sites, simply changing the values at the top to match the settings for the site it is in. To get information from our database, create or modify our databases, or create or change records, we use the same exact queries that we did from the command prompt. Our only difference is that we do it through a function, and store the response in a variable:

```
1. $results = $mysql->query("select * from albums");
```

The `$results` variable here, like our connection variable, is a reference. It is a pointer that lets us continue to communicate with the database and specifies what we are looking for, but is not the actual data. To get the data, we need to ask for each record. Since our example is very small, we will get all of the results at once, and build an array:

```
1. while($row = $results->fetch_assoc()){
2.     $data[]=$row;
3. }
```

This block of code uses the while statement to get every record available in our result set. Note that we used the variable with our result pointer to get the results, and not our connection itself. Inside our loop, we are simply taking the row (in this case, each album from our albums table) and adding it to a new array called data.

Secured Login

Logging into a web page involves receiving user input, sanitizing and validating their submission, appending any salts, hashing the submission, and sending it to the database for verification. If the database responds that the user's credentials match what is stored we can then continue and create a cookie and/or session so the user can interact with secured content.

Creating a User:

```
1. <?php
2. //create our salt
3. $salt=^%r8yuyg;
4. //store the filtered, salted, hashed version of the password
5. $passwordHash = sha1(filter_var($_POST['password'],$salt, FILTER_SANITIZE_STRING));
6. //Add the user to the database
7. $sql = 'INSERT INTO user ($username, passwordHash) VALUES (?,?)';
8. $result = $db->query($sql, array($_POST['username'], $passwordHash));
9. ?>
```

Logging them in:

```
1. <?php
2. //Prep their login credentials
3. $passwordHash = sha1(filter_var($_POST['password'],$salt, FILTER_SANITIZE_STRING));
4. $sql = 'SELECT username FROM user WHERE username = ? AND passwordHash = ?';
5. $result = $db->query($sql, array($_POST['username'], $passwordHash));
6. //This time, look at the result to see if they exist
7. if ($result->numRows() < 1){
8. echo 'Sorry, your username or password was incorrect!';
9. }
10. else{
11. // Create the session
12. $session_start();
13. $_SESSION['active'] = true;
14. echo ('Welcome back!');
15. }
16. ?>
```

Dynamic Canvas

By adding a loop in a canvas drawing of a circle, and using the random number function in the math library of JavaScript, we can instruct the browser to draw circles (in this example, 1000) of random shapes, sizes, and colors, all over our canvas. Each time the page is loaded or refreshed in the browser, the circles will be redrawn, and since we are using random values, this means our image will change each time.

```
1. <canvas id="myCanvas" width="600" height="600"></canvas>
2. <script>
3. var canvas = document.getElementById("canvas");
4. var ctx = canvas.getContext("2d");
5. var w = canvas.width, h = canvas.height; //Set variables of the width, height of the canvas
6. var i = 0;
7. do {
```

```
8. ctx.fillStyle = "rgb(" + Math.round(255*Math.random()) + "," // Creates an R value
9. + Math.round(255*Math.random()) + "," // ... and for G
10. + Math.round(255*Math.random()) + ")"; // ... and for B
11. ctx.beginPath();
12. ctx.arc(w*Math.random(), h*Math.random(), // creates our random size
13. 50*Math.random(),
14. 0, Math.PI*2, true); // Uses Pi*2 to make the arc a circle
15. ctx.closePath();
16. ctx.fill();
17. } while (++i != 1000); // Loops this do 999 more times
18. </script>
```

Table of Results

If we only need to display the information to the user on the screen, or do not plan on manipulating the data or using the data elsewhere on our page, creating the array of results can be a wasteful use of memory. Instead, we will modify our while loop to create a table of our results:

```
1. <table width="75%">
2. <tr><th>Title</th><th>Artist</th><th>Year</th><tr>
3. <?php
4. while($row = $results->fetch_assoc){
5. echo "<tr><td>$row[title]</td><td>$row[artist]</td><td>$row[year]</td></tr>";
6. }
7. ?>
8. </table>
```

This approach only stores one record at a time in our \$row variable and is much more conservative, especially when using larger data sets. You will notice we nested our PHP within regular html in this example. Take a look at what our whole page might look like, assuming we also created our database.php file:

```
1. <?php
2. require("database.php"); // Now that this file is loaded, we can use $mysql on this page
3. $query = "select title, artist, year from albums";
4. $results = $mysql->query($query);
5. ?>
6. <table width="75%">
7. <tr><th>Title</th><th>Artist</th><th>Year</th><tr>
8. <?php
9. while($row = $results->fetch_assoc){
10. echo "<tr><td>$row[title]</td><td>$row[artist]</td><td>$row[year]</td></tr>";
11. }
12. ?>
13. </table>
```

At this point, we now have all of the data we need from the database, and since we will not need the database anymore for this example, we can close our connection by adding the following after our table:

```
1. <?php mysql->close($mysql); ?>
```

Repopulating Forms

If a user has submitted a form or is editing existing data, we can use these values to repopulate our form so the user does not have to type them in again. This is done by echoing the value inside the quotes in the value attribute of the form element. For example, if our name was in the URI as page.php?name=myName, we could make it the value of the input field with:

1. `<form action='page.php' method='get'>`
2. `<input type='text' value='<?php echo $_GET['name']; ?>' />`
3. `</form>`

By suppressing errors with `error_reporting`, using this technique, and with a little logic, we can combine all of the elements of providing the form, validating the submission, and taking action on what the form is for, all from one single page.

With some careful planning, we can provide our user with a form, check their submission, and store their information, all from one page. First we need to create some pseudo-code of the flow of logic so we can dictate what we want to do, and under what conditions:

1. `<?php`
2. `if(form has not been submitted){`
3. show user the blank form
4. `}`
5. `else{`
6. check the form for errors
7. `if (there are errors){`
8. show user their form and data
9. `}`
10. `}`
11. `?>`

By following the logical order of events, the above pseudo-code represents what would be a perfectly functional page. With a couple of tweaks, however, we can make some improvements. First, in this example, we would have to create the same form twice—once blank, and again with placeholders for what the user submitted. While we could use copy/paste and then modify the second copy, this will greatly inflate the overall size of our page. We can also simplify our logic by reversing the order of events in our code. First, we will see if the form has been submitted. If it has, we will check for errors. If there are none, we will complete the submission. After that, we will simply display the form with placeholders for user submitted data if there are errors or the form has not been submitted. This will cover both cases in one place and replaces our if/else and nested if with three if statements:

1. `<?php`
2. `if(form as been submitted){`
3. check it for errors.
4. create a status flag declaring if there are errors or not
5. `}`
6. `if(the status flag is set to "ok"){ // The form must have been submitted, and there are no errors`
7. submit the user info to the database
8. send any confirmation emails
9. display a success message
10. set the status flag to show the form is complete
11. `}`
12. `if(the status flag is anything other than "ok" or does not exist){ //either there were errors, or the form has not been submitted`
13. show the form with placeholders for submitted data
14. `}`
15. `?>`

To make this form more flexible, we can declare an array of field names in our first if statement that lists what elements in our table are required or need to be validated. Once we have done this, we can check each submitted field to see if it needs to be checked, pass it through the appropriate tests, and create a list of feedback. There are a number of ways to approach this. Here we will create an array of responses as our error flag. If there are no errors, we will simply set it to “OK.” We will create a hidden field with a known value that will be included every time the form is submitted. This will help deter outside scripts from using our form, as they would need to know to include the hidden field (which we will check for in our logic) in order for our script to respond. In our example, we will make a short registration form:

```
1. if($_GET[hiddenDate]==now() [check]){
2. $check = array('firstName', 'lastName', 'email', 'email2'); //We will require these fields
3. foreach($check as $field){ [make sure & is in php section] [include "for each thing in" way to remember
4. if($field in $_GET) [verify]{
5. //sanitize variable
6. if(length < 3){ //establishes that our required fields should be at least 3 characters long
7. $_GET[$field]=""; //clear the user submitted value as it does not qualify
8. $errors[]="$field is required and must be at least 3 characters long";
9. }
10. }
11. }
12. if($_GET['email'] != $_GET['email2']){ // Make sure the user entered the same email twice
13. $errors[]="Both email fields must match";
14. $_GET['email']=""; $_GET['email2']="";
15. }
16. else{ // email wasn't entered and/or fields matched
17. if(!empty($_GET['email'])){ // Eliminate the possibility that email is simply empty
18. if(validate($_GET['email'], EMAIL)==false{ $errors[]="Invalid email address" [check]; // We only need to validate one, since
    they are the same
19. $_GET['email']=""; $_GET['email2']="";
20. }
21. }
22. if(!isset($errors)){ // if nothing tripped an error, the array was never created
23. $errors='ok'; // Set the flag to 'ok' (or 1, or whatever else you like) so next section fires
24. }
25. }
```

We have now checked all of the required fields from our form. We may have had more fields, but for whatever reason are not concerned enough about their contents to require or validate them. You may be wondering why we are validating in PHP since JavaScript can do this before the form is submitted in real time. There are two reasons (re)validating with PHP can help you. First, is the more obvious case in which the end user's browser, firewall network policy, etc. has disabled JavaScript. While your form would still function, you would end up with possibly invalid data. Second is that any bots that find your form and attempt to use it maliciously are likely to read your form's destination and send data directly to your processing script, also circumventing your validation. This allows you to add sanitization through PHP in addition to other safety precautions to further harden your site.

Next we will take a look at some options of what we can do once the form has been checked and is OK. Ultimately, there are two main tasks for you here. The first, is do whatever it is you want to do with what the user gave you—email the results, store them in a database, process a registration or login, use them to search data or affect output, etc. The second, is to provide feedback that demonstrates to the user how their action affected the system. It could be search results, a “successful” notice like “Thank You for Registering!” or the result of their interaction, like them being logged into the system.

```
1. if($check=='ok'){ // email ourselves a copy of what they submitted and tell them they are done
2. mail("us@oursite.com", "$_GET[firstName] created and account.", print_r($_GET,true), "From: noreply@oursite.com");
3. echo "Thank you for registering!";
4. }
```

Finally, our last logical test will be true if the user has not submitted anything or if there were errors. By creating this section as follows, we can support both cases at the same time:

```
1. if($errors!='ok'){ //there were errors or the form is not submitted ?>
2. foreach($errors as $error){echo "$error<br>";}
3. <form action='<?php echo $_SERVER['PHP_SELF']; ?>' method='get' name='registration'>
4. <input type='text' name='firstName' value='<?php echo $_GET['firstName']; ?>' /><br/>
```

```
5. <input type='text' name='lastName' value='<?php echo $_GET['lastName']; ?>' /><br/>
6. <input type='text' name='email' value='<?php echo $_GET['email']; ?>' /><br/>
7. <input type='text' name='email2' value='<?php echo $_GET['email2']; ?>' /><br/>
8. <input type='submit' name='Register' value='submit' />
9. </form>
10. <?php } ?>
```

In our last section, the foreach in the second line will print any errors that were added to the array. Since we have reached this point, \$errors either is an array and our entries will print to the screen, or it was never set, and will not show anything on the screen if we are suppressing notices. If you want to avoid the notice generated when the form has not been submitted, we could wrap line 2 with an if statement:

```
1. if(!empty($errors)){foreach($errors as $error){echo "$error<br>";}}
```

In our form you will see we re-entered PHP inside of the value attribute of each input. By echoing the value of the input in our get array, if there is one, we will re-populate our form with what the user entered. Since the first section of our code already checked these values if the form was submitted, any bad entries will have already been reset to nothing, helping the user see what needs to be re-entered.

This effectively completes our one page form, validation, and response. We could add jQuery validation on top of our form elements to improve the user experience as well by validating during the form completion process, but bear in mind this is a progressive enhancement, meaning we should assume JavaScript is off, and that anything we use that works improves upon an already working system.

Drag and Drop

Certain tags in HTML5 now support the ability to be treated as drag and droppable items. Items that support the ability allow for attributes including draggable, ondragenter, ondragover, ondragstart, ondragend, and ondrop. When we want to define the actions that take place when one of these conditions is met, we need to call a JavaScript function, that we define ourselves. We will look at our example by creating it in layers, first defining the structure with HTML, then adding our CSS apply our visual, and finally we will add our JavaScript to give it full functionality.

The first piece of our structure is to define the places in our page where moveable objects are allowed to be. These will typically represent the start and end areas that we are allowed to move objects to and from, like a product page to a shopping cart icon, or just two big empty areas. We will create a simple two location page for now. To define our two areas that are drag and drop friendly, we define our divs as we are accustomed to doing and simply add the references to actions that are allowed, or that we want to instigate actions or changes in our visual cues:

```
1. <div id="startingLocation" ondragenter="return dragenter(event)" ondragover="return hover(event)" ondrop="return
   drop(event)"> </div>
2. <div id="endingLocation" ondragenter="return dragenter(event)" ondragover="return hover(event)" ondrop="return
   drop(event)"> </div>
```

Next, we will add the objects we want to interact with. They need a place to live when the page loads, so we will put them in the startingLocation div.

```
1. <div id="startingLocation" ondragenter="return dragenter(event)" ondragover="return hover(event)" ondrop="return
   drop(event)">
2. <div id="item1" draggable="true" ondragstart="return start(event)" ondragend="return end(event)">Item #1</div>
3. <div id="item2" draggable="true" ondragstart="return start(event)" ondragend="return end(event)">Item #2</div>
4. <div id="item3" draggable="true" ondragstart="return start(event)" ondragend="return end(event)">Item #3</div>
5. </div>
```

While this now gives us a drag and drop foundation, it is not exactly user friendly yet. If you save and test what we have, you will find a very blank screen that is probably rather difficult to interact with as we cannot tell where the different objects start and end, and even at that we have no actions. To address this, we need to add some CSS to our file:

```
1. <style type="text/css">
2. #startingLocation, #endingLocation{
3. Float:left;
4. Width:200px;
5. Height:200px;
6. Margin:10px;
7. }
8. #startingLocation{
9. Background-color:red;
10. }
11. #endingLocation{
12. Background-color:green;
13. }
14. #item1, #item2, #item3{
15. Width:60px;
16. Height:60px;
17. Padding:5px;
18. Margin:10px;
19. }
20. </style>
```

To give us functionality, we need to add JavaScript to dictate what happens when items are moved around on the screen. We need to provide the start function permission to move items, dictate what information it needs to bring with the dragged object, and what to display when the object is in motion:

```
1. <script type="text/javascript">
2. function start(event){
3. //Give the draggable object permission to move
4. event.dataTransfer.effectAllowed='move';
5. //Grabs the dragged items ID for reference
6. event.dataTransfer.setData("id",event.target.getAttribute('id'));
7. // Sets our drag image with no offset
8. event.dataTransfer.setDragImage(event.target, 0, 0);
9. return true;
10. }
11. </script>
```

Next, we need to define what happens to our objects when they are held over an area that takes drops. To do this, we will add the definition of the hover() function we referred to when we created our HTML:

```
1. function hover(){
2. //reads the ID we provided of the dragged item
3. var iddraggable = event.dataTransfer.getData("id");
4. // reads the ID of the object we are hovering over
5. var id = event.target.getAttribute('id');
6. //All items can be dropped into endingLocation
7. if(id=='endingLocation') return false; else return true;
8. }
```

If we wanted to declare that only our first two draggable items are allowed into the endingLocation box, we would change our if statement to specify which items are allowed:

```
1. If(id=='endingLocation')&& (iddraggable=='item1' || iddraggable=='item2') return false;
```

Next we need to complete the act of moving our item to its new location. We will add one more function we have already made reference to in our HTML, `drop()`:

```
1. function drop(){
2.   var iddraggable = event.dataTransfer.getData('id');
3.   event.target.appendChild(document.getElementById(iddraggable));
4.   event.stopPropagation();
5.   return false;
6. }
```

Finally, we need to clean up. Now that our item is dropped, we do not need to worry about its value any longer:

```
1. function end(){
2.   event.dataTransfer.clearData('id');
3.   return true;
4. }
```

If we were going to use our drag and drop system as a shopping cart, we would want to flesh out more actions in our `end` function. We would add code to add the item to the session or cookie record of our shopping cart, and could trigger other actions like updating our cart total on the screen or prompting for a number of that item we want in our cart.

This page titled [3.2: Integration Examples](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [5.2: Integration Examples](#) by [Michael Mendez](#) has no license indicated.

3.3: Finishing Touches

Search Engine Optimization (SEO)

Search engine optimization is the process of making your site the best possible candidate for favorable listing placement in search engine results. The factors that weigh in on a site's scoring and ranking are constantly growing and evolving, and encompass far greater than just the correspondence between a searched word or phrase against the page content in your site.

Covering all aspects of SEO is a task in and of itself just for one search engine. Accounting for the differences between Google, Yahoo, Bing, and all the others would be a text in and of itself and would be out of date before the printer finished the first page.

True optimization is an almost daily task, involving monitoring changes in algorithms, refining the site as content changes, and using systems like Google's Ad Sense for commercial placements on search results pages. Even at that, portions of today's SEO algorithms are out of your control, in that they take into account things that are out of your control like previous traffic, outside links, and more. Start with these basics first to get a good start:

1. Make sure you are mobile optimized
2. Use meta tags and `<h>` tags to emphasize important content
3. Use those same keywords as page titles
4. Do not include too many keywords in your meta tags
5. Update your content regularly
6. Integrate with social media for exposure

Analytics

A close cousin to the tasks of optimizing your site for search engines is optimizing it for your target audience. There is no better way to do this than to understand who your audience is, which you can find in your website logs. You can learn surprising things with analytics, which has spurred many of the extreme large data sets now in existence, and resulted in some of the more controversial features of websites like tailoring ads to your interests or recent searches.

Within your web server's logs, you might find that your visitors are coming from countries you had not anticipated—signs that may induce you to add additional language support, improving their experience. You might find that your users are trying to use mobile devices that you were not quite supporting, or that they are using a resolution higher than you thought, allowing you to redesign your site to provide more content or a smaller interface.

Tracking where a user goes, what they look at, how long they look at it. In fact, every action they take can provide insight. Aggregating this data across a volume of users over time give you the means to discover things you (and perhaps even your users) are not aware of.

You can see these data sets in use when you see features like “Other users also viewed this show” or “You recently looked at this item.” Following up on trends by interviewing users has also shown companies how to convert more leads. To address users leaving with items in their shopping cart before purchasing, some companies discovered through surveys that most of those users decided against a purchase because of shipping costs, or uncertainty over taxes and surcharges. This allowed them to make changes to their site, providing more information sooner to address concerns, and allowed for particular marketing techniques. Instead of making all shipping free, a number of companies will trigger automatic follow-up emails when digital shopping carts are abandoned, or orders are immediately canceled, by offering free shipping or additional coupons or discounts to convince the shopper to complete the sale, improving their conversion rate.

To begin with analytics, you can start by taking a look at the raw logs your server creates. Since our example throughout the book has been using a LAMP, your log location is probably in `/etc/httpd/logs`. A file search for the word logs will likely reveal the correct folder for your particular installation. Depending on your server configuration this folder may have one or more files that track things like people accessing files on your site (everything from pages to images and anything else a user can see or use) as well as errors that were encountered or reported by elements of your server involved in rendering pages.

These files by themselves are not easy to interpret in aggregate form until you have spent time working with them or are seeking answers that are small amounts of the record like the last error or searching for a particular file. Instead, most people who interact

with logs prefer to use an outside program that reads the files and helps them see trends over time so they can extrapolate more information than the raw file provides by itself. A free, open source solution (for personal use) that you can use to this end is AWStats, which compiles your logs into a variety of charts, graphs, and tables that focus on a number of aspects of your site like where visitors are coming from, what browser they used, what pages are most popular, and more.

For more specialized information, many sites create databases that track everything about their site, a user's entire experience, and apply everything else they know about a user to determine larger questions like how best to get the customer to return to the site again or what coupons to offer them like the shopping cart example above.

Privacy Statement

You may wish to include (and abide by) a privacy statement for your users. Spell out exactly what type of information you collect about them, how long it is stored, whether or not you will share or sell their information to another party, and so on. This is also a good place to spell out or link to a location where they can request that their information be removed as well.

This type of statement will allow users to determine if they are comfortable using your site and gives transparency into how you will treat their data.

Terms of Use

In much the same vein as a privacy statement you may also wish to include a terms of use that clarifies to the user the extent to which they are allowed to use your site and its information. We are all familiar with the epic novel terms of use documents that are frequently included with software and hardware purchases these days, but there is a growing movement to embrace a more non-lawyer friendly set of these types of documents.

Which one you decide to use is up to you, and there a great deal of examples in the wild. This is one area in which you may still want to include a lawyer, attorney, or your company's legal department before posting.

This page titled [3.3: Finishing Touches](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- [5.3: Finishing Touches](#) by [Michael Mendez](#) has no license indicated.

3.4: Now What?

By this point you are hopefully comfortable with the fundamentals of web design, and have the ability to balance and integrate at least those languages covered within in order to optimize your site and provide a positive, useful experience for your end users. You have, as some might say, just enough knowledge to be dangerous. There is still far more to learn. Each topic we covered in this text was a tip of the iceberg. The languages we covered, methods and approaches, design techniques, histories and all go into much greater depth.

This text took effort to give you an understanding of each topic, but in many implementations each of these topics is addressed by different parties. Fortune 500 companies typically have one or more employees addressing focused topics such as networking, database development, front-end programming, etc. The consumer research and design aspects alone have created entire departments. Who your team is composed of and how many of you there are become determined by the size of the project, the size of your company, the particular needs and complexities of the project in questions, as well as time line and budget. Similarly sized, timed, and budgeted projects may have distinctly difference staffing simply because one is a financial system requiring input from accountants and lawyers and the other is a museum, requiring input from librarians and art historians.

Now it is time to get some real projects under your belt. After a few full implementations, you will learn which aspects are of most interest or come most naturally to you. Once you have these identified, you have found your niche. Then it is time to go deeper, learning the facets and minutia of one or more of the topics we covered. The references section is a treasure trove of excellent resources to go deeper into any of the topics covered here.

1 <http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf>

2 www.nist.gov

3 www.dhs.gov/topic/cybersecurity

4 www.us-cert.gov

5 php.net/manual/en/filter.filters.sanitize.php

6 www.php.net/manual/en/filter.filters.validate.php

This page titled 3.4: Now What? is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Andrea Barrios](#).

- 5.4: Now What? by [Michael Mendez](#) has no license indicated.

Index

A

Absolute path

[1.5: Navigation](#)

Address

[1.4: Text Layout](#)

analytics

[3.3: Finishing Touches](#)

Anchors

[1.15: Font and Text Decoration](#)

Article

[1.4: Text Layout](#)

Aside

[1.4: Text Layout](#)

attributes

[1.3: Page Layout](#)

audio

[1.10: Media Support](#)

B

background

[1.14: Layout Formatting](#)

Base path

[1.5: Navigation](#)

Bezier curve

[1.9: Canvas](#)

body

[1.3: Page Layout](#)

Borders

[1.14: Layout Formatting](#)

Box model

[1.14: Layout Formatting](#)

Button

[1.4: Text Layout](#)

C

Calendar options

[1.8: Forms](#)

Canvas

[1.9: Canvas](#)

Caption

[1.4: Text Layout](#)

Cite

[1.4: Text Layout](#)

classes

[1.13: Rule Structure](#)

Credentials

[3.1: Security](#)

CSS

[1.1: Markup Languages](#)

[1.13: Rule Structure](#)

css layers

[1.14: Layout Formatting](#)

css layout

[1.14: Layout Formatting](#)

D

Definition lists

[1.4: Text Layout](#)

Div

[1.3: Page Layout](#)

Document Object Model

[1.1: Markup Languages](#)

Document Type

[1.2: Creating HTML Files](#)

Drag and drop

[3.2: Integration Examples](#)

Drawn text

[1.9: Canvas](#)

Dynamic Canvas

[3.2: Integration Examples](#)

E

entities

[1.4: Text Layout](#)

Execution Functions

[3.1: Security](#)

F

Favicon

[1.6: Graphics](#)

figure

[1.4: Text Layout](#)

File format

[1.2: Creating HTML Files](#)

float

[1.14: Layout Formatting](#)

Footer

[1.3: Page Layout](#)

Form fields

[1.8: Forms](#)

Forms

[1.8: Forms](#)

G

Get

[1.8: Forms](#)

Graphic formats

[1.6: Graphics](#)

H

HASHES

[3.1: Security](#)

Head

[1.3: Page Layout](#)

header

[1.3: Page Layout](#)

Headings

[1.15: Font and Text Decoration](#)

HTML5

[1.1: Markup Languages](#)

I

IDs

[1.13: Rule Structure](#)

J

JavaScript Security

[3.1: Security](#)

L

linking

[1.5: Navigation](#)

M

Margins

[1.14: Layout Formatting](#)

Mark

[1.4: Text Layout](#)

Meter

[1.4: Text Layout](#)

Mobil devices

[1.11: Mobile Device Support](#)

Mouse cursor

[1.14: Layout Formatting](#)

MySQL Connecting

[3.2: Integration Examples](#)

MySQL Security

[3.1: Security](#)

N

Nav

[1.4: Text Layout](#)

O

Order of precedence

[1.13: Rule Structure](#)

Ordered lists

[1.4: Text Layout](#)

P

Padding

[1.14: Layout Formatting](#)

page formatting

[1.14: Layout Formatting](#)

Paragraphs

[1.4: Text Layout](#)

PHP Security

[3.1: Security](#)

Positioning

[1.14: Layout Formatting](#)

Post

[1.8: Forms](#)

Privacy Statement

[3.3: Finishing Touches](#)

Progress

[1.4: Text Layout](#)

R

Raster graphics

[1.6: Graphics](#)

Rectangles

[1.9: Canvas](#)

Relative path

[1.5: Navigation](#)

Repopulating forms

[3.2: Integration Examples](#)

risk

[3.1: Security](#)

S

Search Engine Optimization

[3.3: Finishing Touches](#)

Secured Login

[3.2: Integration Examples](#)

Segregated Systems

[3.1: Security](#)

Selectors

[1.13: Rule Structure](#)

SEO

[3.3: Finishing Touches](#)

slicing

[1.6: Graphics](#)

span

[1.3: Page Layout](#)

Spanning

[1.7: Tables](#)

Styling attributes

[1.12: Tags to Avoid](#)

T

tables

[1.7: Tables](#)

Tags

[1.3: Page Layout](#)

Tags to avoid

[1.12: Tags to Avoid](#)

target

[1.5: Navigation](#)

Terms of Use

[3.3: Finishing Touches](#)

Text layout

[1.4: Text Layout](#)

Text styles

[1.15: Font and Text Decoration](#)

Time

[1.4: Text Layout](#)

Time options

[1.8: Forms](#)

Triangles

[1.9: Canvas](#)

U

Unordered lists

[1.4: Text Layout](#)

V

Vector graphics

[1.6: Graphics](#)

video

[1.10: Media Support](#)

Visually impaired considerations

[1.15: Font and Text Decoration](#)

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: CIS111-24171 (ONL): Info Systems-Program/DB Mgmt

Webpages: 36

All licenses found:

- [Undeclared](#): 88.9% (32 pages)
- [CC BY 4.0](#): 11.1% (4 pages)

By Page

- [CIS111-24171 \(ONL\): Info Systems-Program/DB Mgmt - Undeclared](#)
 - [Front Matter - Undeclared](#)
 - [TitlePage - Undeclared](#)
 - [InfoPage - Undeclared](#)
 - [Table of Contents - Undeclared](#)
 - [Licensing - Undeclared](#)
 - [1: Document Markup - Undeclared](#)
 - [1.1: Markup Languages - Undeclared](#)
 - [1.2: Creating HTML Files - Undeclared](#)
 - [1.3: Page Layout - Undeclared](#)
 - [1.4: Text Layout - Undeclared](#)
 - [1.5: Navigation - Undeclared](#)
 - [1.6: Graphics - Undeclared](#)
 - [1.7: Tables - Undeclared](#)
 - [1.8: Forms - Undeclared](#)
 - [1.9: Canvas - Undeclared](#)
 - [1.10: Media Support - Undeclared](#)
 - [1.11: Mobile Device Support - Undeclared](#)
 - [1.12: Tags to Avoid - Undeclared](#)
 - [1.13: Rule Structure - Undeclared](#)
 - [1.14: Layout Formatting - Undeclared](#)
 - [1.15: Font and Text Decoration - Undeclared](#)
 - [1.16: Responsive Styling - Undeclared](#)
 - [2: CSS and Styling a Web Page - CC BY 4.0](#)
 - [2.1: Web Page Header and Footer - CC BY 4.0](#)
 - [2.2: Adding the form to the Page - CC BY 4.0](#)
 - [2.3: Exercises - CC BY 4.0](#)
 - [3: Trying It Together - Undeclared](#)
 - [3.1: Security - Undeclared](#)
 - [3.2: Integration Examples - Undeclared](#)
 - [3.3: Finishing Touches - Undeclared](#)
 - [3.4: Now What? - Undeclared](#)
 - [Back Matter - Undeclared](#)
 - [Index - Undeclared](#)
 - [Glossary - Undeclared](#)
 - [Detailed Licensing - Undeclared](#)