



TASK

HTML Primer

[Visit our website](#)

Introduction

WELCOME TO THE WEB DEVELOPMENT OVERVIEW TASK!

In this task, you'll learn the basics of the web, including where it originated and some technical details about how it works, and you'll also learn how to develop a web application that runs on the internet. As you progress from someone who *uses* web applications to someone who actually *designs and builds* them, we will peel back the layers of the internet to show you what really lies beneath the surface.

You're going to delve into HyperText Markup Language (HTML), a markup language that all software developers need to have at least a basic understanding of, learning to use it to create a basic static web page. Let's get started!

THE WORLD WIDE WEB

What is the World Wide Web, really? It is a global information system consisting of web pages linked to each other using **hyperlinks**. These are the links that allow us to navigate from one page on a website to another. They also allow us to navigate to pages from other websites from around the world. It is this linking technology that creates the effect of an infinite web of information that we navigate daily.



A note from the
HyperionDev Team

Even though we can't imagine our lives without it, the World Wide Web is a rather recent invention. It was invented by Tim Berners-Lee, an English computer scientist, in 1989. Since its invention, it has expanded exponentially until it has become intricately interwoven into every part of our lives. Our work, entertainment, communication, and even our culture is strongly influenced by this powerful technology.

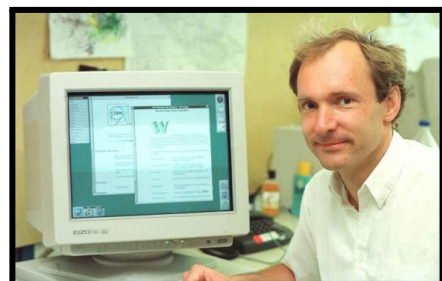


Figure 1: Tim Berners-Lee ¹

CORE COMPONENTS OF THE WORLD WIDE WEB

The World Wide Web consists of three key components. The first is called a **Universal Resource Locator**, or URL. This is a unique identifier assigned to each page and resource on the web. It allows us to identify and retrieve the specific page or video/audio/etc. file that we want over the internet.

The second technology is the language used to create web pages. As you know, this language is called **HyperText Markup Language (HTML)**. Unlike other programming languages that allow us to create programs that actually perform tasks, this language is used to create the format of the page – what the contents are and how they are placed on the page. **HyperText** is a way of organising information that lets you jump easily between different pieces of content. Instead of reading in a straight line, like in a book, you can click on links to move to related topics or pages.

The third technology is the protocol used to request and transfer web pages from one location to another. The internet is a frantically busy and complex medium of communication, and in order for us to ensure that we transfer things successfully from one location to another, we must have rules of transfer – a protocol – for devices to adhere to. This protocol is called **HyperText Transfer Protocol**, better known as **HTTP**.

HOW THE WEB WORKS

We all access the web using a **client**. A client can be a phone, laptop, desktop, etc. – basically anything we can use to access the web. To get to a particular resource (web page, etc.) on the web, we often open a **browser** and use a **URL** to specify what we want to see.

¹ (2009, October). History of the Web. *World Wide Web Foundation*.
<http://webfoundation.org/about/vision/history-of-the-web/>

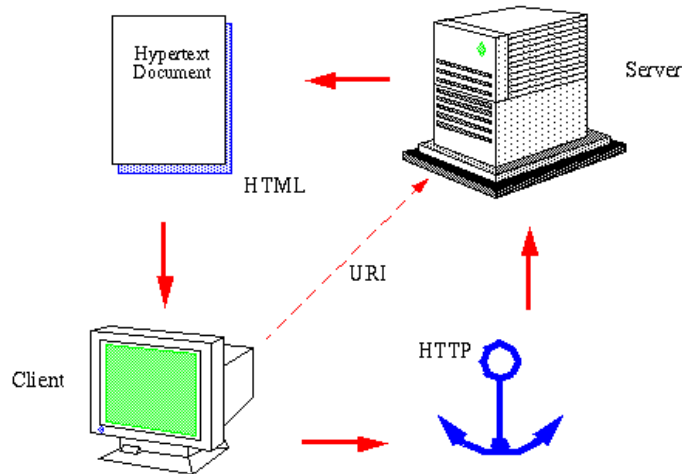
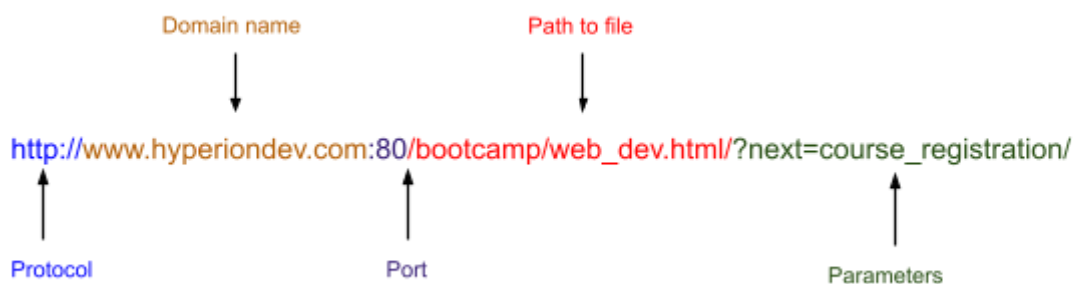


Figure 2: Frystyk, H. (1994, July). *The World-Wide Web* [Illustration]. W3C.
<https://www.w3.org/People/Frystyk/thesis/WWW.html>

A **web browser** is an application program on your web-accessing device that allows you to view websites. Chrome, Firefox, and Microsoft Edge are examples of web browsers. A browser takes a URL, which is the address of the website you want to visit, as input.

A URL is a type of **Uniform Resource Identifier**, or **URI**, which identifies the resource by specifying its location on the web. A URI is a way to give a unique name or address to resources on the internet, and a URL is a specific type of URI. This URL is a human-friendly address for a particular resource (for example, a page called **index.html**) on a particular web server somewhere in the world.

Consider the following fictional URL:



As you can see, the URL contains a lot of information:

1. It identifies the protocol being used to send information. In the example above, the protocol being used is HTTP.
2. It identifies the domain name of the web server on which the resource can be found, e.g. `www.hyperiondev.com`.

3. It identifies the port on the server. In this example, the port number is given as port 80. In reality, if the default HTTP ports are used (port 80 is the default for HTTP, port 443 for HTTPS), they don't have to be given in the URL.
4. It gives the path to the resource on the web server, e.g. /bootcamp/web_dev.html
5. Parameters can be passed using the URL. Parameters are passed as key-value pairs (?key=value&key2=value2), e.g. ?next=course_registration

A web server is a computer that is set up to store and share many web resources, including the HTML files you will create, along with any images, videos, and CSS that you add to your HTML page. The function of the browser is to locate the server specified by the URL.

The browser will send an **HTTP** request for the web page to the server that stores it. The server receives requests from all over the world and sends an HTTP response with the requested resource back to the client's browser. After the server sends the page to the browser, the browser is then able to render the page on the screen for the user to view.

Web servers don't just contain static HTML pages. Back-end systems have been created due to the demand for more dynamic and responsive interaction with web applications. **Back-end** development has to do with writing code that sits on the server and dynamically builds resources that will be returned to the client. This code can be written using programming languages like F#, Python, etc. Many back-end applications interact with databases that store large amounts of data.

THE HTTP REQUEST-RESPONSE CYCLE

The HTTP request-response cycle is the fundamental process that occurs when a client communicates with a server over the web. This cycle involves a series of steps where the client sends a request to the server, and the server responds accordingly. The cycle provides us with an understanding of the way information flows through the web.

1. The user provides a client with a URL.
2. The client then builds a **request** for information.
3. The server receives this request and uses it to build a **response** that contains the requested information.
4. The response is sent back to the client in the requested format to be rendered by the client.

HTTP Request

An HTTP request is a message sent from a client to a server to make a request for a particular resource or to perform a specific action. HTTP (HyperText Transfer Protocol) is the foundation of any data exchange on the web. The most common types of HTTP requests are GET and POST.

The first line in the HTTP request method is referred to as the **request line**, and the following lines are called the **header lines**. The request line consists of three fields: the method field, the URL field, and the HTTP version field. For subsequent use cases, the HTTP 1.1 version will be utilised. See the example below:

```
GET /somedir/page.html HTTP/1.1
```

The method field can take on several different values:

Method	Description
GET	Used when a browser requests an object specified in the URL.
POST	Often used when a user enters form data, which is requesting a web page from the server. However, the contents of the page are dependent on the data entered by the user.
DELETE	Allows a user or application to delete an object on a web server.
PUT	Creates or replaces a resource with a web server. It is often used by applications that need to upload objects to web servers.
PATCH	Applies partial modifications to a resource without changing the whole data.

Header lines provide additional information about the request or the client making the request. They are key-value pairs separated by a colon. The below example specifies the host on which the object resides, the type of connection, in this case, a non-persistent connection, the browser type of the client and the preferred language of the client.

```
Host: www.someschool.edu
Connection: close
User-Agent: Mozilla/5.0
Accept-language: en
```

The final component of the HTTP request message is the **entity body** or **message body**. This contains data related to the request, typically used in POST, PUT, or PATCH requests. Not all requests have a message body.

HTTP Response

An HTTP response message is sent by a server to a client as a result of an HTTP request made by the client. The response contains information about the status of the request, along with optional data in the message body. The response message format consists of three components: a status line, six header lines, and an entity body.

The status line consists of three fields: the protocol version field, a status code, and a corresponding status message. The example below the status line indicates the server is utilising HTTP/1.1, with a status code of 200 and corresponding message OK, which tells us the server has found and is sending the requested object to the client.

```
HTTP/1.1 200 OK
```

Analysing the header lines below, the first line tells us that the Transmission Control Protocol (TCP) connection to the server is closed after sending the message. The *date* line captures the time the response message was created and sent by the server. The *server* indicates the version of the Apache web server being run by the server. *Last-modified* tells us when the resource object was created or last updated. The *content-length* field indicates the size in bytes of the object being sent. Finally, *content-type* indicates the type of the object being sent in the entity body is HTML text.

```
Connection: close
Date: Tue, 14 Nov 2023 15:35:05 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 10 Oct 2023 14:23:46 GMT
Content-Length: 6821
Content-Type: text/html
```

The entity body contains the requested object itself and is stored in the format specified by the content-type. In this case, the HTML content is JSON data in the entity body.

```
<!DOCTYPE html>
```

```
<html>
<head>
  <title>Example Page</title>
</head>
<body>
  <h1>Hello, World!</h1>
</body>
</html>
```

HTTP Response Status Codes

HTTP status codes are three-digit numbers that are returned by a server in response to a client's request made to the server. These codes are grouped into different classes, each class having a specific meaning. Here are some of the common HTTP status code classes:

1. Informational Responses (100–199):
 - 100 Continue
2. Successful Responses (200–299):
 - 200 OK: The request succeeded, and the information was returned in the response.
3. Redirection Responses (300–399):
 - 301 Moved Permanently: The requested object has been permanently moved, and the new URL is specified in the Location: header of the response message. This will automatically be retrieved by the client.
4. Client Error Responses (400–499):
 - 400 Bad Request: Generic error code indicating that the request could not be understood by the server.
 - 404 Not Found: The requested document does not exist on this server.
5. Server Error Responses (500–599):
 - 505 HTTP Version Not Supported: The requested HTTP protocol version is not supported by the server.

THE EVOLUTION OF THE WEB

The evolution of the web can generally be divided into two phases: Web 1.0 and **Web 2.0**. The first generation of websites were basically one-way communication channels. The author would create a web page with some text and images, primarily to communicate information to customers. There was no means of interacting with the web page; all you could do was browse the pages on the website. Since there was no means of changing the web page which was being

viewed without actually taking it off the internet and editing the HTML, we can think of these web pages as being static. On a Web 1.0 site, the user sits back and consumes the contents. For example, check out this classic Web 1.0 site [here](#).

Web 2.0 is all about allowing the user to interact with and contribute content to the website. This is done by providing some means for the user to enter a comment, upload a picture, or “like” something that has been added by someone else. This transition from passive consumption to active contribution to the content of the web page characterises the evolution of the web from 1.0 to 2.0. Because we can actually change the details of the web pages we access, we can think of these web pages as dynamic. Examples of ways to engage with a dynamic website include:

- posting a comment on someone’s Facebook wall.
- creating a page on Wikipedia or editing one that has already been added.
- creating an investment account online using your bank’s website.

Web 2.0 allows for the personalisation of our user experience for any given site. For example, after we log onto a social networking site like Facebook or X (formerly Twitter), we see information that is specific to our own personal user account. This is possible because, instead of a single pre-written web page being sent to your browser when you request a page, your request is processed by a program behind the scenes. The program then extracts data relating to you, which is then used to personalise your page.

USER ACCOUNTS

This is where the concept of a user account comes in. The program that runs behind the scenes needs to know the person it is dealing with in order to personalise a template specifically for them. Each person who uses a web application has a user account with a unique username that identifies them. When you log in to Gmail, your email address is used to identify you and link you with the personal data that will be presented to you. The user account is a core concept in dynamic web development.

SPOT CHECK 1

Let’s see what you can remember from this section.

1. What are the three key components of the World Wide Web?
2. What are the main differences between Web 1.0 and Web 2.0?



Extra resource

This task has provided a very basic overview of how the web works. There is a lot more to it than that! As you progress through the Bootcamp, your understanding of how the web works will increase. However, if you are interested in a little more detail regarding the fundamental **protocols** and **infrastructure** that make the web work, we highly recommend these additional readings:

1. <https://web.stanford.edu/class/msande91si/www-spr04/readings/week1/InternetWhitepaper.htm>
2. <http://www.cs.kent.edu/~svirdi/Ebook/wdp/ch01.pdf>



A note from the
HyperionDev Team

Check out [this infographic](#) that compares front-end and back-end development.

WHY HTML?

HTML is used for front-end development. Front-end development focuses on providing the user with a good user experience. As a software engineer, your primary job is likely to be working on the back-end of an application. You will write the code that will run on the server and make everything work. You will design and code algorithms to solve problems.

So why are you learning about HTML? For several reasons. First, you will have to work with front-end developers to create applications that aren't just technically great but that users find easy and enjoyable to use. Therefore, it is good to have at least a basic understanding of the tools used by front-end developers. Also, your back-end code may have to use HTML tags to send data to the front-end. Besides, HTML is used in many popular development frameworks such as JavaServer Faces

(JSF), Grails, Django, Ruby, and Express. Given the preceding, all software engineers must have and be able to demonstrate at least a basic understanding of HTML.

INTRODUCTION TO HTML

HTML is a language that we use to write files that tell the browser how to layout the text and images on a page. We use HTML *tags* to define how the page must be structured.

HTML Tags

HTML **tags** are placed on the left and the right of the element you want to markup to wrap around the element.

For example:

```
<opening tag>Some text here.</closing tag>
```

This is the general pattern that we follow for all tags in HTML. There are a few exceptions, which we will discuss later. The words ‘opening tag’ and ‘closing tag’ are just placeholders we use to illustrate the pattern. Instead of those words, we are going to use special keywords or elements that modify the appearance of our webpage.

Note that the opening and closing tags are not the same. The opening tag consists of an opening angled bracket, `<`, the name of the element, and a closing angled bracket, `>`. The closing tag consists of an opening angled bracket, `<`, a forward slash, `/`, then the name of the tag, and finally the closing angled bracket, `>`.

```
<!DOCTYPE html>

<html>
<head>
  <title>My first web page!</title>
</head>

<body>
  <p>I am learning to develop a dynamic web application.</p>
</body>
</html>
```

Example of HTML in a simple text file

The HTML tags indicate to the browser what sort of structure the content is contained in. Note that HTML does not include the *style* of the content (e.g. font, colour, size, etc.), which is done using CSS (Cascading Style Sheets), but only the structure and content itself.

HTML Elements

An element usually consists of an opening tag (`<element_name>`) and a closing tag (`</element_name>`), which contains the element's name surrounded by angle brackets and the content in between:

```
<element_name>...content...</element_name>
```

Example of an HTML element:

```
<p>This element is going to result in this paragraph of text being displayed  
in the browser</p>
```

Try this:

- Double click on the file called **example.html** (in the same Dropbox folder as this task) to open it in the browser.
- Examine how the HTML page renders in the browser.
- Now, right-click in the browser and select the option 'View page source.'

The first Heading

The content in our first paragraph is kept here

And our second paragraph content goes here

Using bold an italics

This is what bold looks like.

This is an example of *italics*

Back	Alt+Left Arrow
Forward	Alt+Right Arrow
Reload	Ctrl+R
Save as...	Ctrl+S
Print...	Ctrl+P
Cast...	
Translate to English	
View page source	Ctrl+U
Inspect	Ctrl+Shift+I

- You will see the HTML used to create this webpage that includes many HTML tags. For example, you will notice the tags shown below:

```
<h2> Using bold an italics </h2>
```

```
<p> <b>This</b> is what bold looks like.</p> <!--This is also a tag and needs to be closed as shown here-->  
<p> This is an example of <em>italics</em> </p> <!--em stands for emphasis, and thus makes it in italics -->
```

When the browser encounters the tag `<h2>` it knows to treat the information between the opening `<h2>` tag and the closing `</h2>` tag as a heading. Similarly, the browser will display the information between the tags `<p>` and `</p>` as a paragraph of text.

- You will learn more about specific HTML elements soon.

BASIC LAYOUT/TEMPLATE OF AN HTML PAGE

A typical HTML document consists of a **doctype**, which indicates which version of HTML to load; a **head**, which contains metadata about the page; and a **body**, containing the actual content.

A general layout template that you can use before even starting to worry about what sort of content you want to display is set out below:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

The doctype is indicated at the top of the page, and when typing 'html' it defaults to HTML5. This is one of the only elements that does not need a closing tag. Note that throughout HTML, capitalisation is very important.

Next, we define what content is to follow within the `<html>` tags (note the closing tag at the bottom). Within this `<html>` element, we introduce two other elements, namely `<head>` and `<body>`. Notice that although each of the tags is located on a separate line, we still have **opening tags** matching their **corresponding closing tags**. Notice how the `<html>` tag wraps around its contents. We use a nested order to structure tags on our web page; this means that tags are contained within other tags, which may themselves contain more tags. Above, the `<html>` tag *contains* the `<head>` and `<body>` tags. It is important to understand how elements are nested because one of the **most frequent mistakes that students make with HTML is getting the order all mixed up**. For example, it would be wrong to have a closing body tag (`</body>`) after a closing html tag (`</html>`) because the body element should be completely contained or nested within the `<html>` element. It should also be noted that white space is ignored by the browser, so you can lay out the physical spacing of the elements as you please.

ATTRIBUTES

Attributes are things that describe the objects created by HTML elements. For example, `<p>This element is going to result in this paragraph of text being displayed in the browser</p>` would result in a paragraph that contains text. This paragraph can be described using various attributes including align, font size, etc.

Consider the following:

```
<title id="myTitle">My first web page</title>
```

In this case, the element is of type **title**. Next, we have an **id**, which is an attribute of the element (**title**), and has a value of "myTitle". Attributes like this are used mainly for CSS and JavaScript. Then there is a closing **>** which indicates that you have finished defining the attributes of the element.

COMMON HTML ELEMENTS

We have already encountered some commonly used elements that are used to create most web pages. Some of these (and some new elements) are summarised below:

- **Titles:** a piece of metadata that should be included in all web pages is the **<title>** element. The **<title>** element:
 - defines a title in the browser tab.
 - provides a title for the page when it is added to favourites.
 - displays a title for the page in search engine results.

As noted before, metadata should be contained in the **<head>** of the HTML document.

Example of a title element:

```
<head>
<title>Portfolio</title>
</head>
```

- **Headings:** as you would with a word document, use headings to show the structure of your web page. This is important because search engines use the headings to index the structure and content of your web pages. There are six heading elements you can use: **<h1>** to **<h6>**, where the **<h1>** element is used for the most important headings and the **<h6>** element for the least important.

Example of a heading element:

```
<h1>Online Portfolio of work</h1>
<h2>About me</h2>
```

- **Paragraphs:** add paragraphs of text using the `<p>` element as follows:

```
<p>This is an example of a paragraph. Paragraphs usually contain more text than headings and are not used by search engines to structure the content of your web page. </p>
```

- **Line breaks:** to do the equivalent of pressing enter to get a line break between text, use the `
` element. This element does not have a matching closing tag. This should make sense because there is no content that you could put within a `
` element. Elements like this, with no content or matching closing tags, are known as *void elements*.
- **Horizontal rule:** this is another void element. By adding the HTML element `<hr>` to your web page, you will create a horizontal rule.
- **Lists:** these can either be **ordered lists** `` or **unordered lists** ``. Ordered simply means that the list is numbered, i.e. 1, 2, 3, etc. and unordered is in the form of bullet points. For lists, keeping track of how far you are with the nesting of the various elements is **very** important. We highly recommend that you use indentations to keep track of what elements fall under what other elements. Remember that indentation and “white space” do not affect the layout of the elements on the web page.

Unordered Lists

In an unordered list, as with most elements, we have to open and close the tags. Within this element, we now want to display some content in our list. This content is inputted in the form of *list items* and thus has the tag ``. So, to create an unordered list with three items in it, we would write it out as follows:

```
<ul>
  <li> Item 1 </li>
  <li> Item 2 </li>
  <li> Item 3 </li>
</ul>
```

Note how the indentation makes the entire structure a lot easier to read. The list items, as seen above, are also closed at the end of the content to indicate to the browser where the content of that specific item ends.

Ordered Lists

Ordered lists work almost the same as unordered lists, except that you use the tag, ``. Instead of showing bullet points, these list items are numbered. Your input list of three ordered items would be created as follows:

```
<ol>
  <li> Item 1 </li>
  <li> Item 2 </li>
  <li> Item 3 </li>
</ol>
```

- **Tables:** these work similarly to lists in terms of nesting elements. First, define the fact that it's a table using the `<table>` tag, and then manually enter the data into the rows. Have a look at the example below:

```
<table>
  <tr>
    <td>Row 1, cell 1</td>
    <td>Row 1, cell 2</td>
    <td>Row 1, cell 3</td>
  </tr>
  <tr>
    <td>Row 2, cell 1</td>
    <td>Row 2, cell 2</td>
    <td>Row 2, cell 3</td>
  </tr>
  <tr>
    <td>Row 3, cell 1</td>
    <td>Row 3, cell 2</td>
    <td>Row 3, cell 3</td>
  </tr>
  <tr>
    <td>Row 4, cell 1</td>
    <td>Row 4, cell 2</td>
    <td>Row 4, cell 3</td>
  </tr>
</table>
```

The table element is defined within the opening and closing tags. Immediately within these tags, there is a *table* row indicated by `<tr>`, which also has a closing tag. Within that first table row, there is a `<td>` tag, which indicates that there is *table data*. A table is shown in the **example2.html** file

so that you can try to correlate what elements contribute to what visual appearance on the web page.

The most important elements for this task can be found in **example.html** and **example2.html**.

HTML SYNTAX

As a Software Engineer, you are going to learn many new languages. Each of these has its own rules, which must be strictly followed for your instructions to be properly processed. The rules of a language are referred to as *syntax*. Examples of common HTML syntax errors include spelling the name of an element incorrectly, or not closing tags properly or in the wrong order. You are bound to make mistakes that will violate these rules, and that will cause problems when you try to view web pages in the browser! We all make syntax errors! Often! Being able to identify and correct these errors becomes easier with time and is an extremely important skill to develop.

To help you identify HTML syntax errors, copy and paste the HTML you want to check into this helpful [tool](#).

LINKS

You can add links to your web page as follows:

```
<a href="url" target="_blank">link text</a>
```

The `<a>` element is used to add all links to a web page. Using this element, you can link to other pages on your website and external web pages, and enable users to send an email.

Linking to other places on your web page

Often, you will want your users to be able to click on a link that will then take them to another part of the same web page. Think about the “back to the top” button – you click on this, and you suddenly view the top of the page again!

To do this, we need to use **id** attributes. An **id** is used to identify one of your HTML elements, such as a paragraph, heading, or table. Then, we can use the link tag to

make the text or image a link that the user clicks on to take them to whichever address we choose!

An **id** can be assigned to any of your elements and is done as follows:

```
<h1 id = "theHeading">My first web page</h1>
```

Notice how the attribute **id** is within the opening tag.

Now that we have this heading, we can look at how to reference it within our text. We use the **<a>** tag, which shows the address we are using. To reference a structure with an **id**, we need to precede the value assigned to the **id** attribute with a **#**, otherwise the browser will think you are looking for a website.

```
<h1 id = "theHeading">My first web page</h1>  
<a href = "#theHeading">Back to top</a>
```

Consider the **example.html** file that contains the elements shown above. If you open it, you will see that it will make the text “Back to top” look like a hyperlink (blue and underlined). When this is clicked, it will take you to the heading with the **id** “theHeading”.

Linking to other web pages

Similarly, we can link to another page. This is done as follows:

```
<a href = "http://www.hyperiondev.com">This cool place!</a>
```

The “**http://**” in front of the address lets the browser know that you are linking to an external website rather than a file on your system.

However, you aren’t limited to creating links through text! All the content that is between the **<a>** tags can be clicked on to get to the destination address.

With the link specified above, if you click on the link, it will change the window you’re currently in. What if you wanted to open the destination address of a link in a new tab? You can add an attribute to the link tag called **target**, which specifies how the link should be opened, e.g. in the same window, new browser instance, or new tab. To open in a new tab, simply modify the link as follows:

```
<a target = "_blank" href = "http://hyperiondev.com" />
  This cool place!
</a>
```

IMAGES

We add images to our website using the `` element, as shown below:

```
<img src =
"http://hyperiondev.com/static/moocadmin/assets/img/hyperiondevlogo.png"
alt="HyperionDev logo" height="150" width="150">
<img src = "images/image1.jpg">
```

Unlike most of the other elements we have explored so far, the `` element doesn't have a closing tag. The `` element has several attributes that define or modify it.

- The **src** attribute points to a URL or a file location. In the example above, the first image uses a URL as the source of an image. The second example shows how the **src** attribute is defined to display an image named **image1.jpg**, which is stored in a folder named *images*. The *images* folder resides in the same folder as your HTML file (web page).
- The **alt** attribute defines the *alternate text* that will be displayed if the image won't display.
- The **height** and **width** attributes define the height and width of the image.



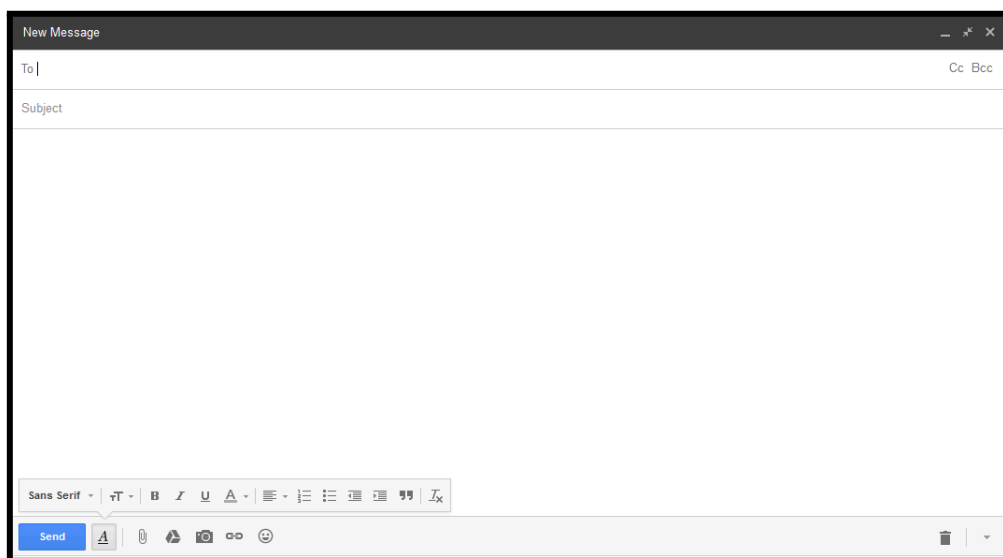
Take note:

When adding images to your web page, it is important to remember that this page may be viewed on many different devices with widely differing screen sizes, resolutions, etc. You want the images to look good independent of the device that is used to view the page. Thus, responsive

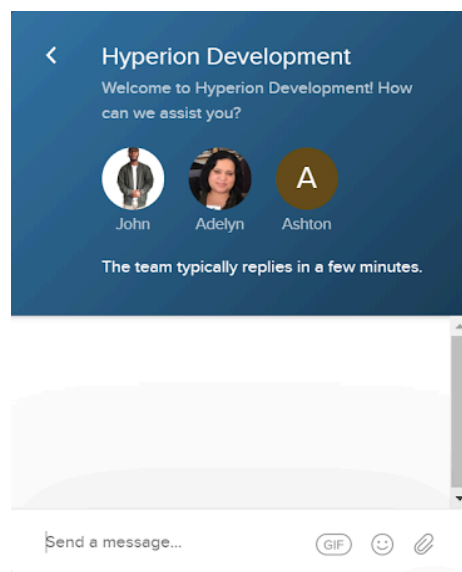
images – images that work well on devices with widely differing screen sizes and resolutions – are important. To learn how to create responsive images, consult the MDN guide about [HTML responsive images](#). Also read Chapter 15 of “HTML5 notes for professionals”, which can be found in the additional reading folder of this task.

HTML FORMS

A dynamic website is driven by user interaction. For users to be able to interact with your website, you need to provide them with the means to enter the information that will be used and displayed on the pages. Forms are the instruments that we use to allow users to enter data in HTML. Forms can be structured in various ways; in fact, web designers often try to make them as cool as possible to encourage users to interact with the site. Here are some examples of different kinds of forms on the Web:

A screenshot of a Gmail 'New Message' form. The form has a dark header bar with the title 'New Message' and window controls. Below the header, there are fields for 'To', 'Cc', 'Bcc', and 'Subject'. The main body of the form is a large, empty text area. At the bottom, there is a rich text editor toolbar with options for font face (Sans Serif), font size, bold, italic, underline, text color, background color, bulleted list, numbered list, link, unlink, indent, outdent, and source code. A 'Send' button is located on the left side of the toolbar, and a trash icon is on the right.

A sophisticated form from Gmail (mail.google.com) – this is the pop-up text editor used to draft an email.

A screenshot of a chat box for 'Hyperion Development'. The chat box has a dark blue header with a back arrow, the title 'Hyperion Development', and a welcome message: 'Welcome to Hyperion Development! How can we assist you?'. Below the header, there are three circular profile pictures of team members: John, Adelyn, and Ashton. Below the profile pictures, it says 'The team typically replies in a few minutes.' The main body of the chat box is a large, empty text area. At the bottom, there is a text input field with the placeholder 'Send a message...' and buttons for adding a GIF, an emoji, and a link.

HyperionDev's (www.hyperiondev.com) chat box is a very sophisticated form, but a form nonetheless.

CREATING A FORM

We won't begin with complex forms like the ones you see above. First, we're going to build a simple form and focus on investigating some of its components. At this stage, our forms won't be functional.

```
<form action = "/action_page.html" method = "get|post">
  <label> First name: </label>
  <input type = "text"><br>
  <label> Surname:</label>
  <input type = "text"><br>
  <label>Gender:</label>
  <select>
    <option value = "male" > Male</option>
    <option value = "female" > Female</option>
    <option value = "other" > Other</option>
  </select>
  <label> Age: </label>
  <input type = "text">
</form>
```

In the example above, we create a form to capture our user's biographical information. It captures the following information:

- First name
- Surname
- Gender
- Age

We expect the user to enter text for their name and surname. We, therefore, use the **input** element. This element has a **type** attribute with the **text** property assigned to it. This displays text boxes in the browser into which users can type input. We add labels to tell our visitors what information we want them to enter into the boxes.

The **select** element is used to create a drop-down menu that users can select from instead of typing out their gender.

Explore a list of other [HTML input types](#) and read Chapter 17 of “HTML5 notes for professionals” in the additional reading folder for more information about input types.

THE METHOD ATTRIBUTE

In the previous example of an HTML form, you would have noticed in the first line of the form element the **method** attribute. The method attribute specifies how the form data is sent to the page specified in the action attribute. There are two methods to send the form data as URL variables using the **GET** method or as an HTTP post-transaction using the **POST** method. These two methods differ in how they pass data to the server in various aspects:

1. Data Submission

- **GET**: Appends data to the URL in the form of query parameters. Data is visible in the URL, and there is a limit to the amount of data that can be sent.
- **POST**: Sends data in the body of the HTTP request. The data is not visible in the URL, and there is typically no practical limit to the amount of data that can be sent.

2. Security

- **GET**: Parameters are included in the URL, making them visible to users and potentially exposing sensitive information. It's not suitable for sensitive data, such as passwords or personal data.
- **POST**: Parameters are included in the request body, making them not visible in the URL. This provides a level of security, especially for sensitive information.

3. Caching involves storing previously retrieved data to enhance performance

- **GET**: Requests can be cached by the browser, and the URL can be bookmarked. It's generally more suitable for idempotent operations, where repeating the request has the same effect.
- **POST**: Requests are typically not cached, and the URL is not bookmarked. It is often used for non-idempotent operations, where repeating the request may have different effects.

4. Idempotency refers to the property where repeating an operation has the same result as performing it once

- **GET**: Generally considered idempotent because multiple identical requests will have the same effect as a single request.
- **POST**: Not necessarily idempotent. Repeating a POST request may have different effects, especially if it involves creating or updating resources.

The choice between GET and POST depends on the nature of the data you are working with, security considerations, and the type of operation you are performing. GET is often used for simple and idempotent operations, while POST is suitable for operations that involve submitting sensitive information or non-idempotent actions.

HTTP GET

An HTTP GET request is a type of HTTP request method used to request data from a specified resource. When a client wants to retrieve information from a server, it sends an HTTP GET request. The GET request is defined by the HTTP protocol, and it consists of the following components:

Request line: The first line of the HTTP GET request is the request line, which includes the method (GET), the target resource (URL or URI), and the HTTP version being used. The format is as follows:

```
GET /login?username=john_doe&password=secretpassword HTTP/1.1
```

In this example, the form data is appended to the URL as query parameters (username=john_doe&password=secretpassword). As mentioned earlier, this is not a recommended practice for sensitive information. If you're dealing with login credentials or other sensitive data, it's better to use HTTP POST requests where the data is included in the request body.

Headers: Following the request line, there can be additional headers that provide information about the request. Headers are key-value pairs separated by a colon. Some common headers in a GET request include:

```
Host: example.com  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
```

Body (optional and rarely used): Unlike the HTTP POST request, the HTTP GET request typically does not include a request body. Data is sent in the URL as query parameters rather than in the request body.

HTTP POST

An HTTP POST request is used to submit data to be processed to a specified resource. Unlike the GET request, which appends data to the URL, the POST request sends data in the body of the HTTP request. This is often used when submitting forms, uploading files, or performing other actions that involve sending data to a server. Here's an explanation of the components of an HTTP POST request:

Request line: The first line of the HTTP POST request is the request line, which includes the method (POST), the target resource (URL or URI), and the HTTP version being used.


```
POST /login HTTP/1.1
```

Headers: Following the request line, there can be additional headers that provide information about the request. Headers are key-value pairs separated by a colon. Some common headers in a POST request include:

```
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
```

The **content-type** header indicates the type of data in the request body. In this example, it's *application/x-www-form-urlencoded*, which is a common format for sending form data. The **content-length** header specifies the length of the request body in bytes.

Request body:

The request body contains the data to be sent to the server. In the case of a POST request, form data or other information is included in the body. The format of the data depends on the content-type specified in the headers. For example, in the case of *application/x-www-form-urlencoded*, the data might be formatted as key-value pairs:

```
username=john_doe&password=secretpassword
```

READABILITY

As you start to create HTML pages with more elements, it becomes increasingly important to make sure that your HTML is easy to read. As you know, in software development, readability is an important principle! Code and markup that are easy to read are easier to debug and maintain than code or markup that are difficult to read.

Indenting your HTML is an important way of improving the readability of your code. For example, consider the HTML below:

```
<!DOCTYPE html><html><head>
<title>My first web page</title>
```

```

</head><body>
<form><label> First name: </label>
<input type = "text"><br>
<label> Surname:</label>
<input type = "text"><br>
<label>Gender:</label><br>
<select><option value = "male" > Male</option>
<option value = "female" > Female</option>
<option value = "other" > Other</option>
</select><br>
<label> Age: </label><br>
<input type = "text"><br>
<input type="submit" value ="Add user">
</form></body></html>

```

The above is perfectly correct HTML that will render properly in the browser, but it is certainly not as easy to read and understand as the code below, which is properly indented:

```

<!DOCTYPE html>
<html>

<head>
  <title>My first web page</title>
  <!--This is a comment, by the way -->
</head>

<body>
  <form>
    <label> First name: </label>
    <input type = "text"><br>
    <label> Surname:</label>
    <input type = "text"><br>
    <label>Gender:</label><br>
    <select>
      <option value = "male" > Male</option>
      <option value = "female" > Female</option>
      <option value = "other" > Other</option>
    </select><br>
    <label> Age: </label><br>
    <input type = "text"><br>
    <input type="submit" value ="Add user">
  </form>
</body>
</html>

```

As you can see above, indentation should be used to show which HTML elements are nested within other HTML elements. As shown above, all the other elements are nested within the <html> element.



Extra resource

Remember that with our courses, you're not alone! To become a competent software developer, it is important to know where to get help when you get stuck. Here you can find resources that provide extra information about [HTML](#).

SPOT CHECK 1 ANSWERS:

1. The Universal Resource Identifier, HyperText Markup Language (HTML), and HyperText Transfer Protocol (HTTP)
2.
 - a. Web 1.0: one-way communication channels where the webpage presented content to the consumer, and the consumer had no way of engaging. These were static web pages.
 - b. Web 2.0: the consumer is able to interact with the website through communication channels like uploads, "liking" and leaving comments. These are dynamic web pages. Dynamic web pages can be personalised to each individual consumer through user accounts.



Take note:

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation.

Give it your best attempt and submit it when you are ready.

You will receive a 100% pass grade once you've submitted the task.

When you submit the task, you will receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer. Take

some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey for this task, which you can use as a self-assessment tool. Please take a moment to complete the survey.

Once you've done that, feel free to progress to the next task.

Auto-graded Task

In this task, you are going to create content for your personal webpage. Don't worry too much about what the webpage looks like at this stage. You will use CSS to add some style and perfect the layout in the next task. For now, focus on the content of the webpage. Does it contain all the information that you would like that introduces you to the world? Strike a balance in your content – this webpage should show more of your personality than a typical CV, but it should still be professional.

Follow these steps:

- Create an HTML page called **index.html**. You can use most code editors to do this, but let's stick with Visual Studio Code.
- On this page, add any elements you would like to create a webpage that acts as an online CV. This is your personal webpage, so feel free to customise it to suit your needs, but make sure that you include the following:
 - A short bio: add a short (no more than three paragraphs) description of yourself. Who are you? What is your experience? What are your passions? What motivates you? What is it that you would most like to do? Etc.
 - Your contact details: e.g. name, contact number, email address, and links to any of your (professional) social media, including LinkedIn.
 - An image of yourself.
 - A list of your skills and competencies.
 - Describe your education.
 - Describe your work experience.

- Incorporate additional functionality using a GET form and a POST form into your CV.
 - Search form (GET):
 - Elements: Include a search box (input type text) that allows visitors to enter specific keywords or terms related to your CV content. Read more on the search box [here](#).
 - Submit button: Add a submit button to initiate the search.
 - Contact form (POST):
 - Elements: Include a form that enables visitors to provide feedback or contact you. This form should contain input fields for the visitor's name, email, and a text area for their message.
 - Submit button: Add a submit button to send the feedback or message.
- Write out examples of request and response messages that include both the header and body for the search, and contact form submission interactions.



Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

[Click here](#) to share your thoughts anonymously.

