Web Development

Muchang Bahng

July 31, 2023

Contents

1 Basics

Basic web development consists of three languages:

1. **HTML** to define the content of web pages
2. **CSS** to specify the layout of web pages
3. **JavaScript** to program the behavior of web pages

Most of vanilla HTML, CSS, and JavaScript can be found on tutorials and online very easily, so I will not repeat all of them here. However, I will include some functionalities that I may use and refer back to often. This will focus mainly on how we can use these languages, along with Python, to gather useful data and implement nice functionalities.

1.1 HTML Commands

1.1.1 Global Attributes and Common Tags

Here is probably the tags that you will see most in an HTML page.

1. The `<main>` tag specifies the main content of the document.
2. The `<article>` tag specifies independent, self-contained content. It does not render as anything special in a browser, but you can use CSS to style the `<article>` element.
3. The `<div>` and `<section>` tags defines a section in a document.

Global attributes are attributes that can be used with all HTML elements.

1. The `id` attribute specifies a unique id for a HTML element. The value of the id attribute must be unique within the HTML document.

```
<h1 id="myHeader">My Header</h1>
```

2. The `title` attribute specifies extra information about an element, shown as a tooltip text when the mouse moves over the element.

```
<p title="Further Information">Information Presented</p>
```

3. The `class` attribute specifies one or more classnames for an element. This is like an id attribute but multiple elements can have the same class name, and an element with multiple classnames are separated with a space.

```
<p class="center large">paragraph</p>
```

4. The `style` attribute specifies an inline CSS style for an element.
5. The `lang` specifies the language of the element's content, and `spellcheck` specifies whether the element is to have its spelling/grammar checked or not.

1.1.2 Images

The `<img>` tag defines an image

```
<img src="image.png" alt="text" style="width;42px;height:42px;">
```

with the following attributes:

1. `src` specifies the source of the image.
2. `alt` specifies the alternative text that will appear if the image fails to render.
3. `width` and height specifies the width and height, respectively.
4. `style` specifies the multiple things defined with CSS
   (a) `vertical-align:top, middle, bottom` aligns the image vertically
   (b) `margin:50px 0px`

1.1.3 Hyperlinks

The `<a>` tag defines a hyperlink, with the following attributes:

1. `href` specifies the URL/file of the destination. Note that this tag may contain absolute URLs, relative URLs, or file names.
2. `target` specifies where to open the linked document. `target=BLANK` opens it in a new tab, while `target=SELF` (default) opens in the current tab.

```
<a href="https://www.linktodestination.com" target="BLANK">Hyperlinked Text</a>
```

To hyperlink an image, simply nest the `<img>` tag in the `<a>` tag.

```
<a href="www.website.com">
    <img src="WebsiteLogo.png" alt="Logo">
</a>
```

1.1.4 Buttons

The `<button>` tag defines a clickable button. You can put text, along with tags like `<img>` inside it. It has multiple attributes:

1. The `type="VALUE"` attribute determines what type of button it is. It takes one of the three values `"button"`, `"reset"`, `"submit"`.
2. The disabled attribute determines whether the button is disabled or not. It is un-clickable, until a JavaScript removes the disabled value.
3. The `onclick="JavaScript"` attribute determines what happens when you click the button.

1.1.5 Audio/Video

The `<audio>` tag is used to embed sound content in a document, such as music or other audio streams. It contains one or more ¡source¿ tags with different audio sources, supporting .mp3, .wav, and .ogg files. The browser will choose the first source it supports. The text between the `<audio>` and `</audio>` tags will only be displayed in browsers that do not support the `<audio>` element.

Attribute	Value	Description
autoplay	autoplay	Specifies that the audio will start playing as soon as it is ready
controls	controls	Specifies that audio controls should be displayed (such as a play/pause button etc)
loop	loop	Specifies that the audio will start over again, every time it is finished
muted	muted	Specifies that the audio output should be muted
preload	auto OR metadata OR none	Specifies if and how the author thinks the audio should be loaded when the page loads
src	*URL*	Specifies the URL of the audio file

```
<audio controls>
  <source src="audio.mp3" type="audio/mpeg">
  Your browser does not support the audio element.
</audio>
```

The `<video>` tag is used to embed video content in a document, such as a movie clip or other video streams. It contains one or more ¡source¿ tags with different video sources. The browser will choose the first source it supports. The text between the `<video>` and `</video>` tags will only be displayed in browsers that do not support the `<video>` element.

```
<video width="320" height="240" controls>
  <source src="movie.mp4" type="video/mp4">
  <source src="movie.ogg" type="video/ogg">
  Your browser does not support the video tag.
</video>
```

Attribute	Value	Description
autoplay	autoplay	Specifies that the video will start playing as soon as it is ready
controls	controls	Specifies that video controls should be displayed (such as a play/pause button etc).
height	*pixels*	Sets the height of the video player
loop	loop	Specifies that the video will start over again, every time it is finished
muted	muted	Specifies that the audio output of the video should be muted
poster	*URL*	Specifies an image to be shown while the video is downloading, or until the user hits the play button
preload	auto OR metadata OR none	Specifies if and how the author thinks the video should be loaded when the page loads
src	*URL*	Specifies the URL of the video file
width	*pixels*	Sets the width of the video player

1.2 CSS Commands

The `<style>` tag is used to define style information (CSS) for a document, placed within the `<head>` tag in the beginning of the document. Inside the `<style>` element you specify how HTML elements should render in a browser.

```
<html>
    <head>
        <style>
            body {background-color:lightblue;}
            h1 {color:red; text-align:center;}
            p {color:blue; font-family:verdana; font-size:20px;}
        </style>
    </head>

    <body>
        <h1>A heading</h1>
        <p>A paragraph.</p>
    </body>
</html>
```

1.2.1 CSS Selectors

The syntax of CSS consists of a selector and a declaration block of the form of a selector, followed by a declaration block (in curly brackets):

```
SELECTOR {PROPERTY:VALUE; PROPERTY:VALUE;}
```

We can select HTML elements based on multiple attributes:

1. (CSS Element Selector) selects HTML elements based on the element name. The CSS rule below will be applied to all HTML element with the `<p>` tag.

   ```
   p {text-align:center; color:red;}
   ```

2. (CSS id Selector) selects HTML elements based on their id. This requires the use of a hash (#) character in the CSS. The CSS rule below will be applied to the HTML element with `id="para1"`.

   ```
   #para1 {text-align:center; color:red;}
   ```

3. (CSS Class Selector) selects HTML elements with a specific class attribute. This requires the use of a period (.) character. The CSS rule below will be applied to all HTML elements with `class="center"`.

   ```
   .center {text-align:center; color:red;}
   ```

4. (CSS Universal Selector) selects all HTML elements on the page. We use an asterisk (*)

   ```
   * {text-align:center; color:blue;}
   ```

5. (CSS Attribute Selector) selects HTML elements with a specific attribute or attribute value. The CSS rule below will be applied to all `<a>` elements with a target attribute, and all `<a>` elements with a `target-"BLANK"` attribute

   ```
   a[target] {background-color: yellow;}
   a[target="BLANK"] {background-color: yellow;}
   ```

Finally, we can group multiple CSS selectors into one as we do below. The CSS rule will be applied to all HTML elements with `<p>` tag, `id="para1"`, or `class="center"`.

```
p, #para1, .center {text-align:center; color:red;}
```

1.2.2 Internal, External, and Inline CSS

There are three ways to apply CSS rules:

1. To import CSS rules to an HTML file from an external source, we can use the `<link>` tag within the `<head>` section and reference the file containing the rules. This is used when wanting to change the look of an entire website, which may have multiple HTML files. The `<rel>` attribute specifies the relationship between the current document and the linked document.

```
<html>
<head>
<link rel="stylesheet" href="mystyle.css">
<head>
<body>

<h1>A heading</h1>
<p>A paragraph.</p>

</body>
</html>
```

   Note that the file must be a CSS file! All the rules can be written in a textedit file as below, and then saved with the .css extension. The file "mystyle.css" should look like this:

```
body {background-color:lightblue;}
h1 {color:navy; margin-left:20px;}
```

2. To apply CSS rules to an HTML file internally, we can just put all the selectors within the `<style>` tag, as we did normally. To save space, only a snippet of the code is shown.

```
<style>
body {background-color:linen;}
h1 {color:maroon; margin-left:40px;}
</style>
```

3. To apply CSS rules to a single element, we use inline styles using the style attribute.

```
<h1 style="color:blue;text-align:center;">This is a heading</h1>
<p style="color:red;">This is a paragraph.</p>
```

If there is more than one style specified for an HTML element, then the CSS style with the highest priority will be applied over the others, with the priority ranking as: inline style, then external and internal style sheets (whichever is applied later within the code), then browser default.

1.2.3 Basic Attributes

We will briefly go over some basic CSS attributes. Thickness and length are usually written in terms of pixels, and modern browsers support 140 color names, along with even more colors in HEX value and RGB format. Some common CSS attributes are:

1. `"width:300px"` is a box-sizing property that keeps the width of the element at 300px.
2. `"background-color:Tomato"`, used for background coloring of texts, sections, etc.
3. `"color:Tomato",` used for coloring of text.

4. `"border: 2px solid Tomato"`, used for formatting borders around an HTML element. We can add additional attributes:
   (a) `"border-style: solid"`
   (b) `"border-width: 25px 10px 4px 35px"` means 25px top, 10px right, 4px bottom, and 35px left.
   (c) `"border-color: red green blue yellow` means red top, green right, blue bottom, and yellow left/.
   (d) `"border-style: dotted solid double dashed"`
   (e) `"border-radius: 5px"` gives you rounded corners.
5. `"opacity:0.3"`, used to edit the opacity/transparency of an element.
6. `"background-image:paper.png"` specifies an image to use as the background of an element. By default the image is repeated to cover the entire element. With this attribute, we can add additional CSS attributes:
   (a) `"background-repeat: no-repeat"` shows the background image only once.
   (b) `"background-position: right top"` is used to specify the position of the background image.
   (c) `"background-attachment: fixed/scroll"` specifies whether the background image should scroll or be fixed.
7. `"margin:25px 50px 75px 100px"` means top margin 25px, right 50px, bottom 75px, left 100px. Marigins are used to create space around elements, outside of any defined borders. "margin:auto;" will horizontally center the element within its container.
8. `"padding: 25px 50px 75px 100px"` means top margin 25px, right 50px, bottom 75px, left 100px. Padding is used to generate space around an element's content, inside of any defined borders.

1.3 JavaScript Commands

Javascript code is always inserted between `<script>` and `</script>` tags. These script tags can be inserted in the `<head>` tag

```
<head>
  <script>
  function myFunction() {
    document.getElementById("demo").innerHTML = "Paragraph changed.";
  }
  </script>
</head>
```

or at the bottom of the body tag (this improves display speed because script interpretation slows down display)

```
<body>

  ...

  <script>
  function myFunction() {
    document.getElementById("demo").innerHTML = "Paragraph changed.";
  }
  </script>
</body>
```

Also, scripts can also be played in external .js files, with obvious advantages of accessibility and organization. For example, the external file myScript.js would contain multiple JavaScript functions (remember, without the script tag).

```
function myFunction1() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}

function myFunction2() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
...
```

and these collections of functions would be called by placing the folllowing script tag with the correct src (source) attribute referencing the .js file.

```
<html>
<body>

<h2>Demo External JavaScript</h2>

<p id="demo">A Paragraph.</p>

<button type="button" onclick="myFunction1()">Try it</button>

<p>This example links to "myScript.js".</p>
<p>(myFunction is stored in "myScript.js")</p>

<script src="myScript.js"></script>

</body>
</html>
```

Finally, scripts can be placed directly in html elements by writing the entire function in the attribute.

```
<button type="button" onclick="document.getElementById("demo").innerHTML = 'Paragraph
changed.'">Try it</button>
```

1.3.1 JavaScript Display

All of these functions introduced below should be written inside a function as below (or in the html tag):

```
function myFunction() {
  ...
}
```

One of the many JavaScript HTML methods is `getElementById()`, which finds an element with a certain id. In most contexts, we write it as `document.getElementById()`.

1. `document.getElementById().innerHTML` changes the element content to anything.

   ```
   document.getElementById("id").innerHTML = "Resulting Text"
   ```

2. `document.getElementById().src` changes the value of the src (source) attribute, which can be used to change images.

```
document.getElementById("id").src = "resulting_image.png"
```

3. `document.getElementById().style.display` changes the CSS display. Setting it equal to block renders the element as a block-level content, none does not dispaly the element at all, flex displays it as a block-level flex box, inherit inherits the display from the parent element, and initial sets the property to its default value.

```
document.getElementById("id").style.display = "block"
```

Some further output functions are below.

1. `window.alert()` pops up an alert box to display data.

```
window.alert("Alert Message")
```

2 Web APIs

When we look at a radio station, there is a power button, some knobs to adjust the volumes, and another knob to adjust the frequency. This is an **interface** that a user can deal with in order to control the machine, while not knowing all *how* these things are done. All the inner mechanisms and wiring within the radio is abstracted away from the user. If we want to graphically show it for more familiarity, we can use a **graphical user interface (GUI)**. An interface that allows us to communicate with a programming application is an **application programming interface (API)**. In here, we will learn how to use API, more specifically web APIs. In web APIs, we use some program to communicate with a complicated web server to have it do commands for us, like give us data or compute something for us. Almost all the time, APIs these days will refer to web APIs.

Even a button on a music playing app is an interface, since when we press the button, it invokes some function that gets the music to start playing. It may be the case the case that our music isn't stored on our phone, but streamed over the internet, and so we must use a web API to retrieve the mp3 data. In fact, it's APIs all the way down. As a user, we don't need to know the implementation or how it works. We just need to know what we're allowed to change. It is also amazing that every browser (e.g. Chrome, Firefox, Safari, Brave, etc.) can run the *same exact* code (HTML, CSS, JavaScript) found over the internet. This works because there is a set of web APIs that web browsers must all implement. The browser is called a **web client** that connects to a **web server**.

2.1 REST API

2.2 HTTP Protocol and Requests

The Python **requests** library allows us to easily make HTTP requests to websites. Note that while requests is useful for getting information from websites, it is not a package for parsing that information, which is done by BeautifulSoup. We can HTTTP request to a website by inputting its URL into the `get` method. It returns the HTTP status code in a `Response` object.

```
import requests

r = requests.get('https://xkcd.com/353/')
print(r)                # <Response [200]>
print(r.status_code)    # 200
```

To see the HTML contents of this page, we can use two functions.

1. `r.text` returns the content of the response in Unicode,which is preferred for textual responses such as HTML or XML.

2. `r.content` returns the content of the response in bytes, which is preferred for image or PDF files.

However, this is essentially a giant string, so this will not help us parse the file for specific attributes or values. They will both look like HTML text in the interpreter, but you can see their main difference in type.

```
print(type(r.text))         <class 'str'>
print(type(r.content))      <class 'bytes'>
```

We can also take images and save them in a file by reading and writing them in bytes.

```
r = requests.get('https://imgs.xkcd.com/comics/python.png')

with open("comic.png", "wb") as f:
    f.write(r.content)
```

Finally, we can look at important general information about the site by calling the `headers` attribute. This tells us the following:

1. *Server* : which server the site is running on
2. *Content-Type* : type of file
3. *Last-Modified* :

```
print(r.headers)

{'Connection': 'keep-alive', 'Content-Length': '90835', 'Server': 'nginx', 'Content-
Type': 'image/png', 'Last-Modified': 'Mon, 01 Feb 2010 13:07:49 GMT', 'ETag':
'"4b66d225-162d3"', 'Expires': 'Sun, 09 Jul 2023 23:05:36 GMT', 'Cache-Control':
'max-age=300', 'Accept-Ranges': 'bytes', 'Date': 'Mon, 10 Jul 2023 01:41:39 GMT',
'Via': '1.1 varnish', 'Age': '0', 'X-Served-By': 'cache-iad-kcgs7200062-IAD', 'X-
Cache': 'HIT', 'X-Cache-Hits': '1', 'X-Timer': 'S1688953299.092258,VS0,VE77'}
```

3 Web Scraping

Web scraping is very useful for gathering data, setting up crawlers, and for NLP tasks.

3.1 BeautifulSoup

3.2 Scrapy

3.3 LXML

4 Front-End Frameworks

The three most popular front-end, client-side web frameworks are React, Angular, and Vue, in decreasing order of popularity. We will use React.

5 Django

Django itself is web framework used for developing web applications based on python. Which is used for making development process more simple and easy. To get a high level overview of Django, we use the *MVT* design pattern:

1. *Model* : The model is the data you want to present in your website, which is usually stored in a database first, usually in a relational database like SQL. Located in `models.py`.

2. *View* : The view is the request handler that returns the relevant template and context based on the request from the user. It is basically a function that takes HTTP requests as arguments, imports the relevant models, and finds out which data to send to the template. Located in `views.py`.
3. *Template* : The template is a text file (e.g. HTML file) containing the layout of the web page, with logic on how to display the data. The templates of an application is located in a directory called `templates`.
4. *URL* : Django also provides a way to navigate around the different pages in a website. Then a user requests a URL, Django decides which view it will send it to. This is done in a file called `urls.py`.

When you have installed Django and created your first Django web application, and the browser requests the URL, this is basically what happens:

1. Django receives the URL, checks the `urls.py` file, and calls the view that matches the URL.
2. The view, located in `views.py`, checks for relevant models.
3. The models are imported from the `models.py` file.
4. The view then sends the data to a specified template in the template folder.
5. The template contains HTML and Django tags, and with the data it returns finished HTML content back to the browser.

We can get started by making a new virtual environment (I used conda) and installing django with pip.

```
(web) mbahng@xps15:~/Desktop$ pip install django
```

We can cd into our directory and use the command

```
(web) mbahng@xps15:~/Desktop$ django-admin startproject tennis_club
```

to initialize a project folder called `tennis_club` in your current working directory. In fact we can look at its tree once it is initialized.

```
(web) mbahng@xps15:~/Desktop$ cd tennis_club/
(web) mbahng@xps15:~/Desktop/tennis_clubj$ tree
.
 tennis_club
    asgi.py
    __init__.py
    settings.py
    urls.py
    wsgi.py
 manage.py

1 directory, 6 files
```

Now to run the default server, we can run the `manage.py` file with the `runserver` keyword.

```
(web) mbahng@xps15:~/Desktop/tennis_club$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply
the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
July 09, 2023 - 19:50:46
Django version 4.2.3, using settings 'tennis_club.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

And now, if we go to `http://127.0.0.1:8000/` or `localhost:8000`, we should see the default layout of our page.

Now we can proceed to built an **app**, which is a web application that has a specific meaning in your project, like a home page, a contact form, or a members database. I will create an app called `members` in this project with the command

```
(web) mbahng@xps15:~/Desktop/tennis_club$ python manage.py startapp members
```

This will create a `members` directory in your project.

5.1 URLs

5.2 Models

Ideally, we would like to work with things stored in databases and extract them based off some request. We will focus on constructing these databases. As mentioned in the beginning, in Django, data is created in objects, called Models, and is actually tables in a database.

5.2.1 Instantiating a Model

Say that we want to create a database in our members app. We navigate to `members/models.py` and instantiate a subclass of `models.Model`. We also want to describe the features of each element.

```
from django.db import models

class Member(models.Model):
  firstname = models.CharField(max_length=255)  # first name
  lastname = models.CharField(max_length=255)   # last name
```

Both fields `firstname` and `lastname` are text fields with a maximum length of 255 characters (this is similar to SQL). This database, along with all other databases, will be stored in the `db.sqlite3` database file in the root of the project. Then we want to propagate these changes by migrating them through the two commands:

```
(web) mbahng@xps15:~/Desktop/tennis_club$ python manage.py makemigrations members
Migrations for 'members':
  members/migrations/0001_initial.py
    - Create model Member
(web) mbahng@xps15:~/Desktop/tennis_club$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, members, sessions
Running migrations:
  Applying members.0001_initial... OK
```

5.2.2 Editing Data in a Model

Now that we have a new database, we can use the Python interpreter to add, update, and delete members to it. We can see that there is no data/queries in it.

```
(web) mbahng@xps15:~/Desktop/tennis_club$ python manage.py shell
Python 3.11.4 (main, Jul  5 2023, 13:45:01) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from members.models import Member
>>> Member.objects.all()
<QuerySet []>
```

1. We can add by instantiating a `Member` object and calling the `save` method.

   ```
   >>> member = Member(firstname='Emil', lastname='Refsnes')
   >>> member.save()
   >>> Member.objects.all().values()
   <QuerySet [{'id': 1, 'firstname': 'Emil', 'lastname': 'Refsnes'}]>
   ```

   To add multiple objects we can simply use for loops.

2. The returned object all all queries supports indexing so you can access them by index and update/save their information.

   ```
   >>> x = Member.objects.all()[0]
   >>> x
   <Member: Member object (1)>
   >>> x.firstname
   'Emil'
   >>> x.firstname = "Stale"
   >>> x.save()
   >>> Member.objects.all().values()
   <QuerySet [{'id': 1, 'firstname': 'Stale', 'lastname': 'Refsnes'}]>
   ```

3. To delete an element, we call the `delete` method.

   ```
   >>> x.delete()
   (1, {'members.Member': 1})
   >>> Member.objects.all().values()
   <QuerySet []>
   ```

5.2.3 Updating a Model

Say that we want to add (or remove) some fields/features in our model. We can just go into `models.py`, make our changes:

```
from django.db import models

class Member(models.Model):
  firstname = models.CharField(max_length=255)
  lastname = models.CharField(max_length=255)
  phone = models.IntegerField()
  joined_date = models.DateField()
```

and make the migration to tell Django to update the database. They will ask us what we want to do with the newly added fields of the already existing data in the prompt below.

```
(web) mbahng@xps15:~/Desktop/tennis_club$ python manage.py makemigrations members
It is impossible to add a non-nullable field 'joined_date' to member without specifying a default. This
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
 2) Quit and manually define a default value in models.py.
Select an option:
```

Some data are by default not allowed to be null values, but we can override this by setting the `null=True` parameter.

```
from django.db import models

class Member(models.Model):
  firstname = models.CharField(max_length=255)
  lastname = models.CharField(max_length=255)
  phone = models.IntegerField(null=True)
  joined_date = models.DateField(null=True)
```

After which we can run the migrate again.

```
(web) mbahng@xps15:~/Desktop/tennis_club$ python manage.py makemigrations members
Migrations for 'members':
  members/migrations/0002_member_joined_date_member_phone.py
    - Add field joined_date to member
    - Add field phone to member
(web) mbahng@xps15:~/Desktop/tennis_club$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, members, sessions
Running migrations:
  Applying members.0002_member_joined_date_member_phone... OK
```

We can simply update the null data as before, in the updating data part in the previous subsection.