Startup Engineering
Balaji S. Srinivasan
Stanford University

Class: class.coursera.org/startup-001
Signup: coursera.org/course/startup
Forums: bit.ly/startup-eng-forum

An introduction to HTML/CSS/JS

A webpage is a program

It's useful to think of a webpage as similar to a C program from Stanford's CS106B. Just like you use gcc to compile C programs at the command line, when you navigate to a page in a web browser it performs a kind of realtime compilation step called rendering. The input is raw text like the left side of this page and the output is the clickable GUI interface in your browser.

Once you start thinking of webpages as akin to programs, you begin to realize the level of sophistication of modern web browsers. Not only do browsers need to compile a fairly complicated language (HTML/CSS/JS) almost instantly to create a user-friendly clickable interface, they have to deal with the fact that you as the end-user are essentially *installing* a program every single time you view a page. They also need to deal robustly with noisy network connections, bizarre character encodings, legacy code dating back to the early 1990s, security considerations of all varieties, and many other complexities that most programs can safely ignore. It's thus no exaggeration to say that a web browser is the most complex piece of code on a computer, comparable only to a compiler or operating systems in terms of its complexity. Indeed, Chrome itself is now the core of an operating system (namely Chrome OS).

From HTTP Request to Rendered Page

Emil Stenstrom put together an excellent high-level overview of the rendering process in the context of a modern web application; let's go through an edited and updated version.

1. You begin by typing a URL into address bar in your preferred browser or clicking a link.
2. The browser parses the URL to find the protocol, host, port, and path (example).
3. If HTTP was specified, it forms a HTTP request.
4. To reach the host, it first needs to translate the human readable host into an IP address, and it does this by doing a DNS lookup on the host (video).
5. Then a socket needs to be opened from the user's computer to that IP address, on the port specified (most often port 80 for HTTP).
6. When a network connection is open, the HTTP request is sent to the host. The details of this connection are specified in the 7-layer OSI model (see examples).
7. The host forwards the request to the server software (most often Apache) configured to listen on the specified port.

8. The server inspects the request (most often only the path), and the subsequent behavior depends on the type of site.

    - For static content:
        - The HTTP response does not change as a function of the user, time of day, geographic location, or other parameters (such as HTTP Request headers).
        - In this case a fast static web server like nginx can rapidly serve up the same HTML/CSS/JS and binary files (jpg, mp4, etc.) to each visitor.
        - Academic webpages like http://startup.stanford.edu are good examples of static content: the experience is the same for each user and there is no login.
    - For dynamic content:
        - The HTTP response *does* change as a function of the user, time of day, geographical location, or the like.
        - In this case you will usually forward dynamic requests from a web-server like nginx (or Apache) to a constantly running server-side daemon (like mod_wsgi hosting Django or node.js behind nginx), with the static requests intercepted and returned by nginx (or even before via caching layers).
        - The server-side web framework you use (such as Python / Django, Ruby / Rails, or node.js / Express) gets access to the full request, and starts to prepare a HTTP response. A web framework is a collection of related libraries for working with HTTP responses and requests (and other things); for example, here's the Express request and response APIs, which allow programmatic manipulation of requests/responses as Javascript objects.
        - To construct the HTTP response a relational database is often accessed. You can think of a relational database as a set of tables similar to Excel tables, with links between rows (example database schema).
        - While sometimes raw SQL is used to access the database, modern web frameworks allow engineers to access data via so-called Object-Relational Mappers (ORMs), such as sequelize.js (for node.js) or the Django ORM (for Python). The ORM provides a high-level way of manipulating data within your language after defining some models. (Example models / instances in sequelize.js).
        - The specific data for the current HTTP request is often obtained via a database search using the ORM (example), based on parameters in the path (or data) of the request.
        - The objects created via the ORM are then used to template an HTML page (server-side templating), to directly return JSON (for usage in APIs or client-side templating), or to otherwise populate the body of the HTTP Response. This body is then conceptually put in an envelope with HTTP Response headers as metadata labeling that envelope.
        - The web framework then returns the HTTP response back to the browser.

9. The browser receives the response. Assuming for now that the web framework used server-side templating and return HTML, this HTML is parsed. Importantly, the browser must be robust to broken or misformatted HTML.

10. A Document Object Model (DOM) tree is built out of the HTML. The DOM is a tree structure representation of a webpage. This is confusing at first as a webpage may look planar rather than hierarchical, but the key is to think of a webpage as composed of chapter headings, subsections, and subsubsections (due to HTML's ancestry as a descendant of SGML, used for formatting books).

11. All browsers provide a standard programmatic Javascript API for interacting with the DOM, though today most engineers manipulate the DOM through the cross-browser JQuery library or higher-level frameworks like Backbone. Compare and contrast JQuery with the legacy APIs to see why.

12. New requests are made to the server for each new resource that is found in the HTML source (typically images, style sheets, and JavaScript files). Go back to step 3 and repeat for each resource.

13. CSS is parsed, and used to annotate each node in the DOM tree with style information on how it should render. CSS controls appearance.

14. Javascript is parsed and executed, and DOM nodes are moved and style information is updated accordingly. That is, Javascript controls behavior, and the Javascript executed on page load can be used to move nodes around or change appearance (by updating or setting CSS styles).

15. The browser renders the page on the screen according to the DOM tree and the final style information for each node.

16. You see the webpage and can interact with it by clicking on buttons or submitting forms. Every link you click or form you submit sends another HTTP request to a server, and the process repeats.

To recap, this is a good overview of what happens on the server and then the client (in this case the browser) when you navigate to a URL. Quite a miracle, and that's just what happens when you click a link[1].

Anatomy of a webpage

Let's now drill down a little from the macroscopic context of HTTP requests and responses to get a bit more detail on the components of a webpage: HTML/CSS/JS. If you've ever written a comment on a blog, you're probably familiar with basic HTML tags like `<b>` to bold text

[1]It's useful to think in a bit more detail about what happens when you hit a key to start a job on a remote EC2 machine. Strike a key and the capacitance changes instantaneously, sending a signal down the USB cable into the computer. The computer's keyboard driver traps the signal, recognizes the key it's from, and sends it to the operating system for handling. The OS determines which application is active and decides whether the keystroke needs to be routed over an internet connection. If it is routed, a set of bytes is sent down the CAT-5 cable into the wall socket, abstraction masked via the 7-layer model, and sent to a router. The router determines the closest router which is likely to know the IP of your destination, and sends that packet onwards. Eventually the packet arrives at the destination computer, enters the via ethernet cable, is trapped by the remote OS, interpreted as an enter stroke, sent to the shell, sent to the current program, and executed. Data from that remote machine is then sent back over the cable and the routers to your machine, which takes the packet, uses it to update the display, and tells you what happened on that remote box. All of that happens within an imperceptible fraction of a second.

and <i> to italicize. But to build a proper webpage you need to understand the division of labor between HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JS (Javascript), abbreviated as HTML/CSS/JS.

If you analogize a webpage to a human, you won't go wrong by thinking of HTML as the skeleton, CSS as the skin and clothing, and Javascript as the dance routine that makes the human move (and possibly change clothing). There are many outstanding books and tutorials in this general area, though be warned that w3schools.com is only so-so despite its incredible PageRank. The WebPlatform.org tutorials on HTML, CSS, and JS are useful indexes of tutorials. You can also use htmldog.com to go through HTML and CSS, and Eloquent Javascript for JS.

HTML: Skeleton and Semantics

HTML is a finite set of about 112 elements (aka tags) that are used to specify the structure of a webpage. Each element has attributes, key-value pairs that specify metadata on the element or modify its appearance in the browser. Here's an example with the <abbr> tag for an abbreviation in which four attributes are explicitly specified:

```
<abbr id="sql-abbrev" class="abbrevs" style="color:purple;"
title="Structured Query Language">SQL</abbr>
```

The most basic HTML elements are things like <p> (paragraph), <ul> (unordered list) and <h1> (largest page header), which correspond closely to the kinds of elements you'd use to structure a book or an academic webpage. The reason for this is that HTML is derived from SGML, which was used to do hierarchical mark up on raw text for book formatting. While academic webpages map fairly well to this hierarchical metaphor (a nested directory of chapters, sections, and subsections), many web pages have significant two-dimensional structure. /You will not go wrong if you think of a webpage to first order[2] as a set of non-overlapping rectangular boxes called elements (example)./

Since the advent of HTML5, the major vendors (Google, Apple, Mozilla, Opera, Microsoft) have equipped their browsers to work with new tags like <nav> (navigation links) and <section> (article sections). These tags provide shortcuts and semantic meaning. Search engines can do more with <section> than or even , as the former tag gives some semantic information (i.e. that it is a section of an article) while the tag alone just specifies that it's a box on a webpage, which could be anything from a login form to a container for a profile photo. In addition to these semantic tags, modern browsers understand new tags like <video>, <audio>, and <canvas>, which go well beyond the book metaphor. Here's a simple HTML page with the major elements DOCTYPE, <head>, and <body>:

The DOCTYPE is an incantation that tells the browser what version of HTML is being used. The <head> contains metadata on the page, which in this case includes the <title> and the character encoding (in this case utf-8). And the <body> contains the meat of the page content, which in this case is just a simple header and paragraph. Finally, here is how it renders in the browser.

[2]This is only "to first order", because you can force elements to overlap in the x/y plane by the use of negative margins or to layer on top of each other using the z-index property.

```
<!DOCTYPE html>
<html>

    <head>
     <title>Hello</title>
     <meta charset="utf-8">
    </head>

    <body>
         <h1>Header</h1>
        <p>Hello world!</p>
    </body>

</html>
```

CSS: Look and Layout

As noted above, while academic webpages tend to use unstyled HTML and map well to the hierarchical book metaphor, most websites today employ some degree of two-dimensional layout, coloration, and typographical customizations. CSS allows you to take the same HTML skeleton and clothe it in different fonts, background colors, and layouts. An early demonstration of the power of CSS was CSS Zen Garden, which showed in principle that you could achieve a clean separation between content and appearance. In practice, several of the designs on CSS Zen Garden are actually very layout sensitive, with images micro-optimized for a particular layout or text heading, but the principle is mostly intact.

Let's take our sample webpage and add some extremely basic CSS to a `<style>` tag in the `<head>` section. Below, we're setting the font size of text within the `<p>` element to 14px and the color to red (see it live).

```
<!DOCTYPE html>
<html>

    <head>
        <title>Hello</title>
        <meta charset="utf-8">
        <style type="text/css">
            p {
                font-size: 14px;
                color: red;
            }
        </style>
    </head>

    <body>
         <h1>Header</h1>
        <p>Hello world!</p>
```

```
18      </body>
19
20  </html>
```

Note that `<h1>` appeared large and bold without specifying anything custom in the `<style>` element. This happens because browsers specify different default styles for each element; moreover these styles can be subtly different. People work around browser inconsistencies by using CSS resets, that zero out all the default settings and normalize them to the same values across browsers. Here is what the same example looks like with the *Normalize CSS* option clicked on the left hand side.

In practice it's a lot of effort to build up a reasonably good looking CSS stylesheet, as it involves everything from the layout of the page to the dropshadows on individual buttons. Fortunately there are now excellent CSS frameworks like Bootstrap, which provide built-in grid systems for 2D layout, typography, element styling, and more. Here's that same example again, except now without the Normalize CSS option and with Bootstrap's default CSS instead. Bootstrap is an extremely good way to get up and running with a new website without reinventing the wheel, and can be readily styled with various themes once you get some traffic.

JS: Dynamics and Behavior

Once you've prettified an HTML page with CSS, it's time to make it dance with JS. In the early days of the web, Javascript was used for popping up alert boxes and simple special effects. Today JS is used for things like:

1. Confirming that the text in an email field is a valid email.
2. Creating various input widgets like autocomplete search boxes.
3. Pulling content from remote servers to create dynamic news feeds.
4. Playing games in your browser.
5. Embedding API functionality to provide things like payments or social integration.

The reason JS rather than Python or Haskell is used for these purposes is that a JS interpreter is present in every web browser (Chrome, Firefox, IE, Safari, Opera), and all elements of a webpage can be manipulated with Javascript APIs. Here's an example with the video tag, and here's an example of changing the CSS style on an HTML tag with JS. Furthermore, in addition to being present in web browser ("client-side") environments, extremely good command line ("server-side") implementations of Javascript are now available, like node.js. JS is now also available in databases like MongoDB, and new libraries are constantly ported to JS. JS is, in other words, the "Next Big Language" as prophesied by Steve Yegge. If you only know one language, you should know Javascript in all its manifestations.

Separation of concerns

Any web app will include HTML, CSS, and JS files in addition to server-side code. One of the advantages of node.js is that the server-side code is also in JS.

In principle, you should scrupulously separate your HTML, CSS, and JS, such that you can change any of the three (or more) files that specify a webpage without worrying about the other two. In practice, you will need to develop all three files (.html, .css. and .js) at the same time as significant changes to the structure of a page will often remove `id` attributes or CSS styles that your JS code depends upon.

Sometimes you will have a choice as to whether you implement a particular function in HTML, CSS, or JS, as they are not entirely independent. For example, it's possible to implement buttons via pure CSS or JS, or to put JS invocations within HTML tags rather than using JQuery. When in doubt, err on the side of separation of concerns: you should try to use HTML for structure, CSS appearance, and JS for behavior.

Tools

As you edit HTML/CSS/JS, the single most important thing to learn are the Chrome Developer Tools. You can pull these up by opening up any page and then going to View -> Developer -> Developer Tools (on a Mac, the keyboard shortcut is Command-Option-I). With these tools, you can do the following:

1. Edit the styles and nodes live on a page.
2. Click an element on a page to find it in the markup, or vice versa.
3. Watch network connections and inspect HTTP requests and responses.
4. Execute Javascript to alter the page.
5. Mimic mobile browsers with the User Agent Switcher.

In addition, you should familiarize yourself with jsfiddle (very useful for sharing examples) and reloading tools like Live Reload or Codekit (more).