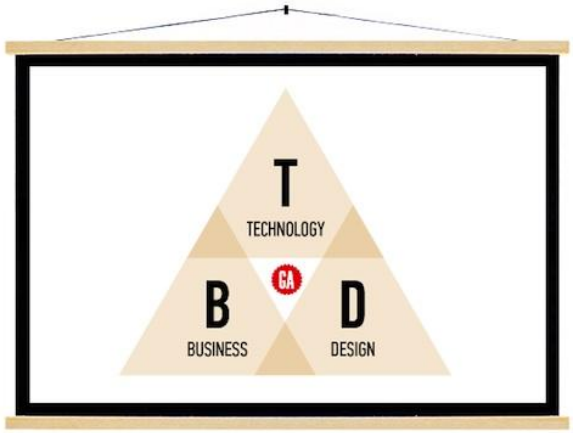
Front End
Web Development

aliasneo@gmail.com

github.com/generaltzou



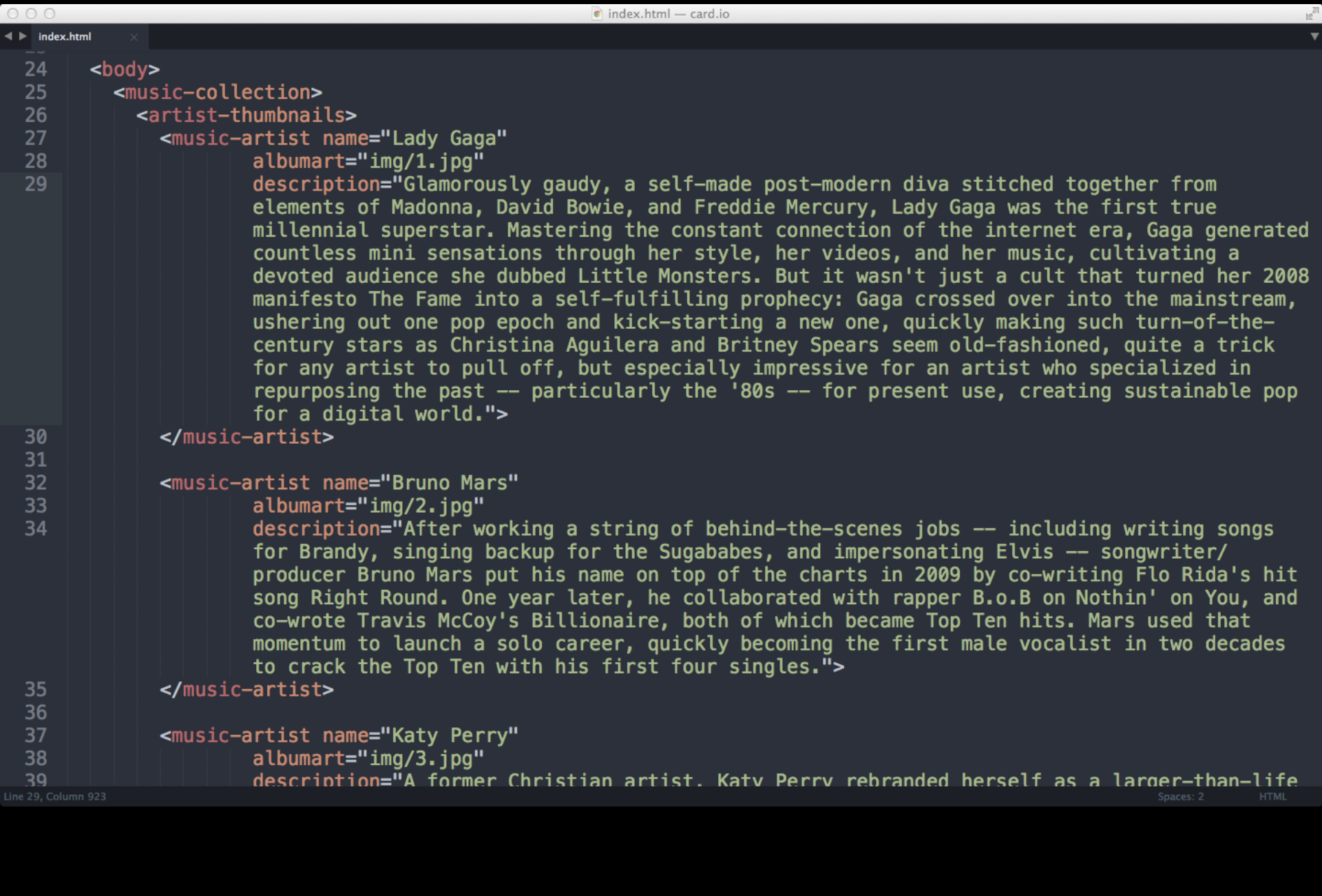
Layer Cake of Technology

User & domain concerns

Machine concerns

```
; =============== S U B R O U T I N E =======================================


sub_40CBC4      proc near               ; CODE XREF: sub_40D250+117↓p
                                        ; sub_40DF04+B4↓p ...

var_14          = dword ptr -14h
var_10          = dword ptr -10h

                push    ebx
                push    esi
                push    edi
                add     esp, 0FFFFFFF8h
                mov     [esp+14h+var_10], edx
                mov     [esp+14h+var_14], eax
                xor     esi, esi
                xor     edi, edi
                mov     eax, ecx
                dec     eax
                test    eax, eax
                jb      short loc_40CC32
                inc     eax
                xor     edx, edx

loc_40CBDF:                             ; CODE XREF: sub_40CBC4+6C↓j
                inc     esi
                and     esi, 0FFh
                xor     ecx, ecx
                mov     cl, ds:byte_41DB1C[esi]
                lea     ebx, [ecx+edi]
                and     ebx, 0FFh
                mov     edi, ebx
                mov     bl, ds:byte_41DB1C[edi]
                mov     ds:byte_41DB1C[esi], bl
                mov     ds:byte_41DB1C[edi], cl
                xor     ebx, ebx
                mov     bl, ds:byte_41DB1C[esi]
                add     ecx, ebx
                and     ecx, 0FFh
                mov     ebx, [esp+14h+var_14]
                mov     bl, [ebx+edx]
                xor     bl, ds:byte_41DB1C[ecx]
                mov     ecx, [esp+14h+var_10]
                mov     [ecx+edx], bl
```

C4: sub 40CBC4


index.html — card.io
index.html
24 <body>
25 <music-collection>
26 <artist-thumbnails>
27 <music-artist name="Lady Gaga"
28 albumart="img/1.jpg"
29 description="Glamorously gaudy, a self-made post-modern diva stitched together from elements of Madonna, David Bowie, and Freddie Mercury, Lady Gaga was the first true millennial superstar. Mastering the constant connection of the internet era, Gaga generated countless mini sensations through her style, her videos, and her music, cultivating a devoted audience she dubbed Little Monsters. But it wasn't just a cult that turned her 2008 manifesto The Fame into a self-fulfilling prophecy: Gaga crossed over into the mainstream, ushering out one pop epoch and kick-starting a new one, quickly making such turn-of-the-century stars as Christina Aguilera and Britney Spears seem old-fashioned, quite a trick for any artist to pull off, but especially impressive for an artist who specialized in repurposing the past -- particularly the '80s -- for present use, creating sustainable pop for a digital world.">
30 </music-artist>
31
32 <music-artist name="Bruno Mars"
33 albumart="img/2.jpg"
34 description="After working a string of behind-the-scenes jobs -- including writing songs for Brandy, singing backup for the Sugababes, and impersonating Elvis -- songwriter/producer Bruno Mars put his name on top of the charts in 2009 by co-writing Flo Rida's hit song Right Round. One year later, he collaborated with rapper B.o.B on Nothin' on You, and co-wrote Travis McCoy's Billionaire, both of which became Top Ten hits. Mars used that momentum to launch a solo career, quickly becoming the first male vocalist in two decades to crack the Top Ten with his first four singles.">
35 </music-artist>
36
37 <music-artist name="Katy Perry"
38 albumart="img/3.jpg"
39 description="A former Christian artist, Katy Perry rebranded herself as a larger-than-life
Line 29, Column 923
Spaces: 2
HTML

Process Innovation

Principles & Patterns

As we build real-world projects together over the next few weeks -- from a responsive web page to a fully interactive mobile-first web application -- we will encounter many technical terms and concepts across HTML, CSS, and JavaScript.

This class is largely project and product-driven. Class time is dedicated to building code together with the instructor and TAs. From the applications we build together in class, we will extract *Principles* and *Patterns* to accommodate beginning learners, or learners who are looking for more structured content outside of code. By referring back to these *Principles & Patterns* as learning objectives while we build our projects in class, we aim to accommodate the diverse range of learner goals in FEWD 32.

Principles

Principles are foundational concepts that underlie computer programming or web development. They are concepts that students should be acquainted with and comfortable articulating to peers and employers in the industry. From the projects we build together in class, we will run across key principles that we highlight here in this deck.

Patterns

Patterns have limited underlying conceptual value but are useful for building up a “toolkit” for students to use in subsequent projects. Nearly all of practical programming is learned by example, taking a simple codebase and modifying it. Examples of patterns include navigational bars, using the various Twitter Bootstrap conventions *xs, sm, md, lg* to lay out content on a grid, and using an IFrame to embed a YouTube video. These types of techniques are learned by copying and pasting, modifying, then reusing the snippets in future projects.

Weeks 1 & 2 Assignment: Responsive Web Page

Principles

- Web applications (High-level overview *HTML, CSS, JavaScript*)
- Structure of an HTML5 web application
- Client/Server Architecture
- The DOM
- Browser Developer Tools
- HTML elements (tags)
- The Box Model for HTML elements
- CSS selectors
    - id and class selectors
    - element selector
- Color
- Layout using **inline** vs. **block** styling
- Element position: static vs. absolute

Patterns

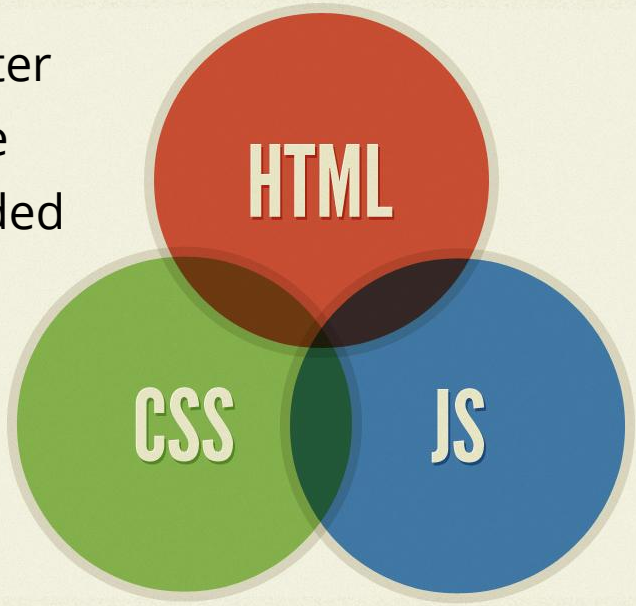
- Navigational header
- Hero image
- Layout of responsive websites using Twitter Bootstrap
    - Containers and Rows
    - 12-column grid across device sizes from extra small (xs) to large (lg)
- .spacer class selector
- Embedded YouTube player using an IFrame
- Swipeable Gallery

Principle

Web Applications

A web application is comprised of HTML, CSS, and JavaScript **files** together with assets (images, video, text metadata) that gets interpreted by a browser.

These files sit on a computer (a server) connected to the Internet and are downloaded from a user's browser.



HTML5 refers to standards that allow you to build web applications that are widely supported and interpreted by web browsers.

Principle

Structure of an HTML5 web app

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <!-- Include CSS dependencies on which your app depends. -->
5    </head>
6    <body>
7      <!-- Your page's content lives here and is comprised of HTML elements/tags. -->
8
9      <!-- Before the body ends is typically a good place to include JavaScript. -->
10   </body>
11 </html>
```

`<!DOCTYPE HTML>` indicates to the browser that we wish to render the page in standards-mode. Other modes are used for old legacy browsers, so stick with standards-mode.

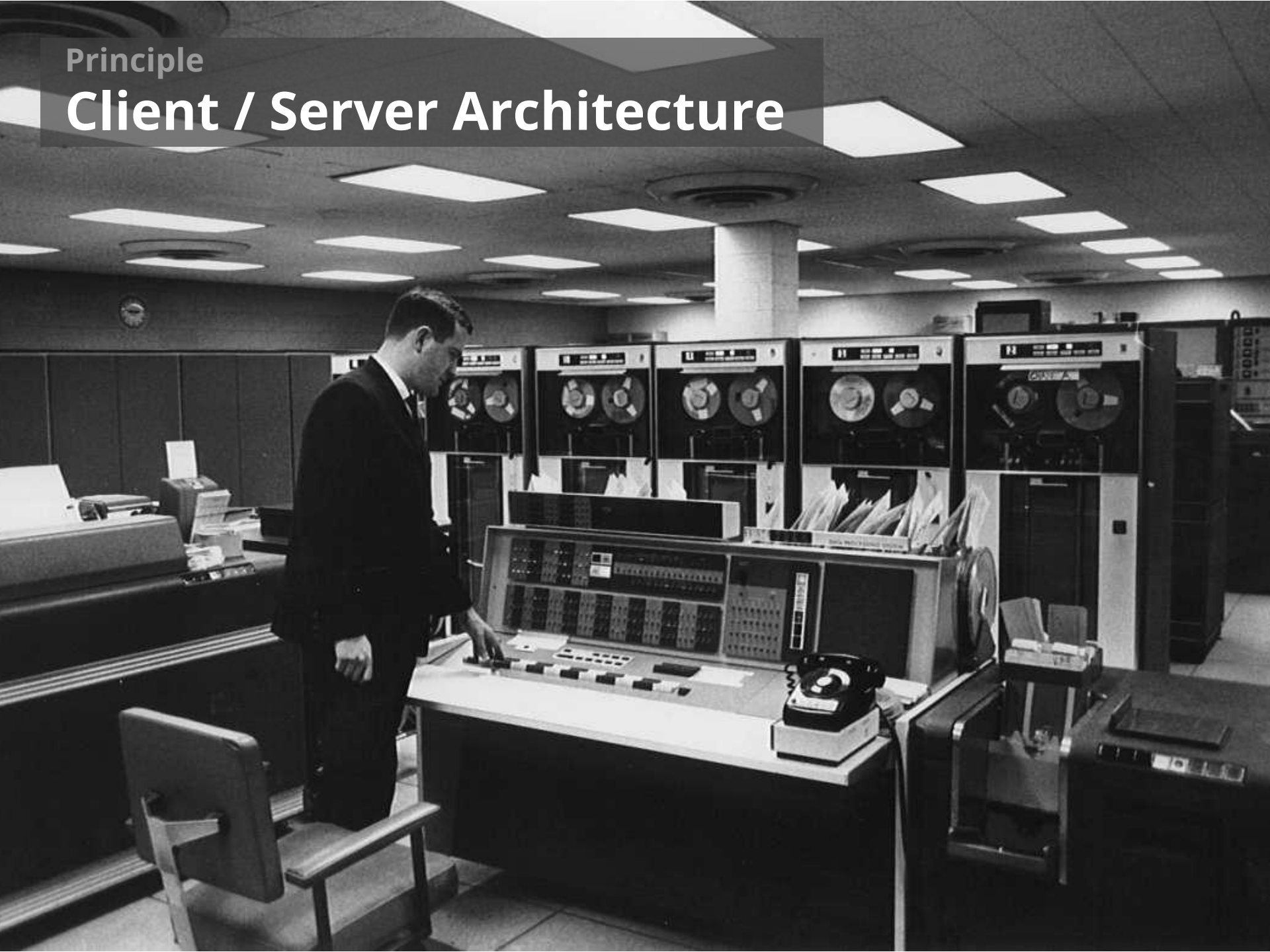
Within the `<head>` section, it is common to include your web page's title such as `<title>300: Rise of an Empire</title>` and also any CSS files on which your HTML depends. Include CSS using the link tag:

```
<link href="style.css" rel="stylesheet">
```

The CSS file could be a local relative path to a file on your computer, or point to any URL on the web. It is common to include external dependencies (such as Twitter Bootstrap CSS) followed by your CSS file that's specific to *your* application.

Principle

Client / Server Architecture



Principle

Client / Server Architecture

In the 80's and much of the 90's, personal computers were not connected to any other computers. You had to manually load software onto it by buying "shrinkwrapped software."

As computers became networked together, the most useful computer applications had a **client / server architecture** where not only did computation happen locally, but some happened remotely.

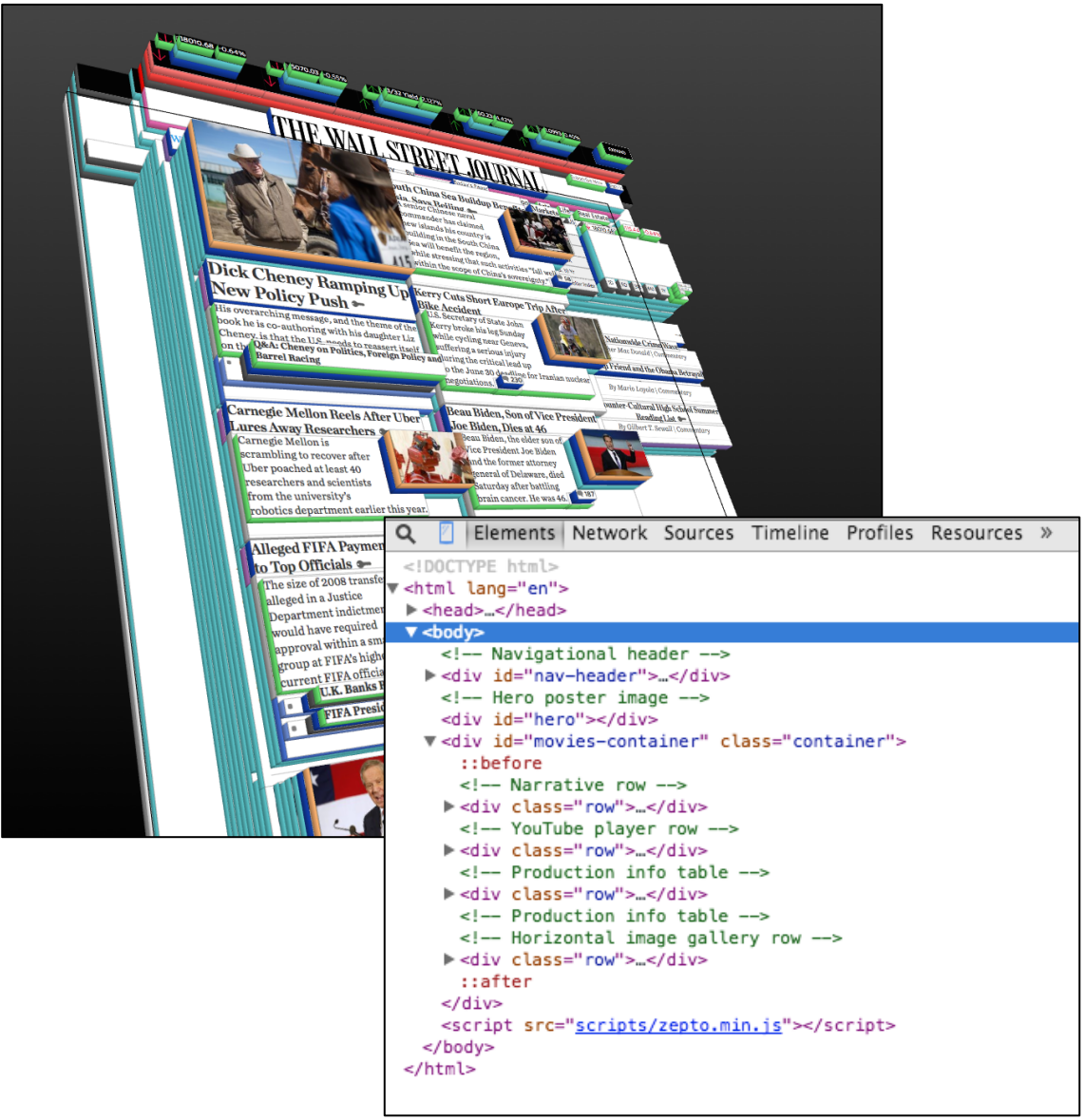
We say clients *request* resources or data from a server. The browser is probably the most famous application you install that uses the client/server model.

Today's most popular clients are browsers on both desktops and mobile devices, and native apps on mobile devices.

As a front-end developer, most of your work is on the client but you periodically need to make requests from the server to fetch data, or to tell the server to do something.

Principle

The Document Object Model



As you've seen, HTML is just text. As this text is sent over the Internet and downloaded by the web browser, it gets "parsed" (read into computer memory) and built into a document structure called the Document Object Model (DOM).

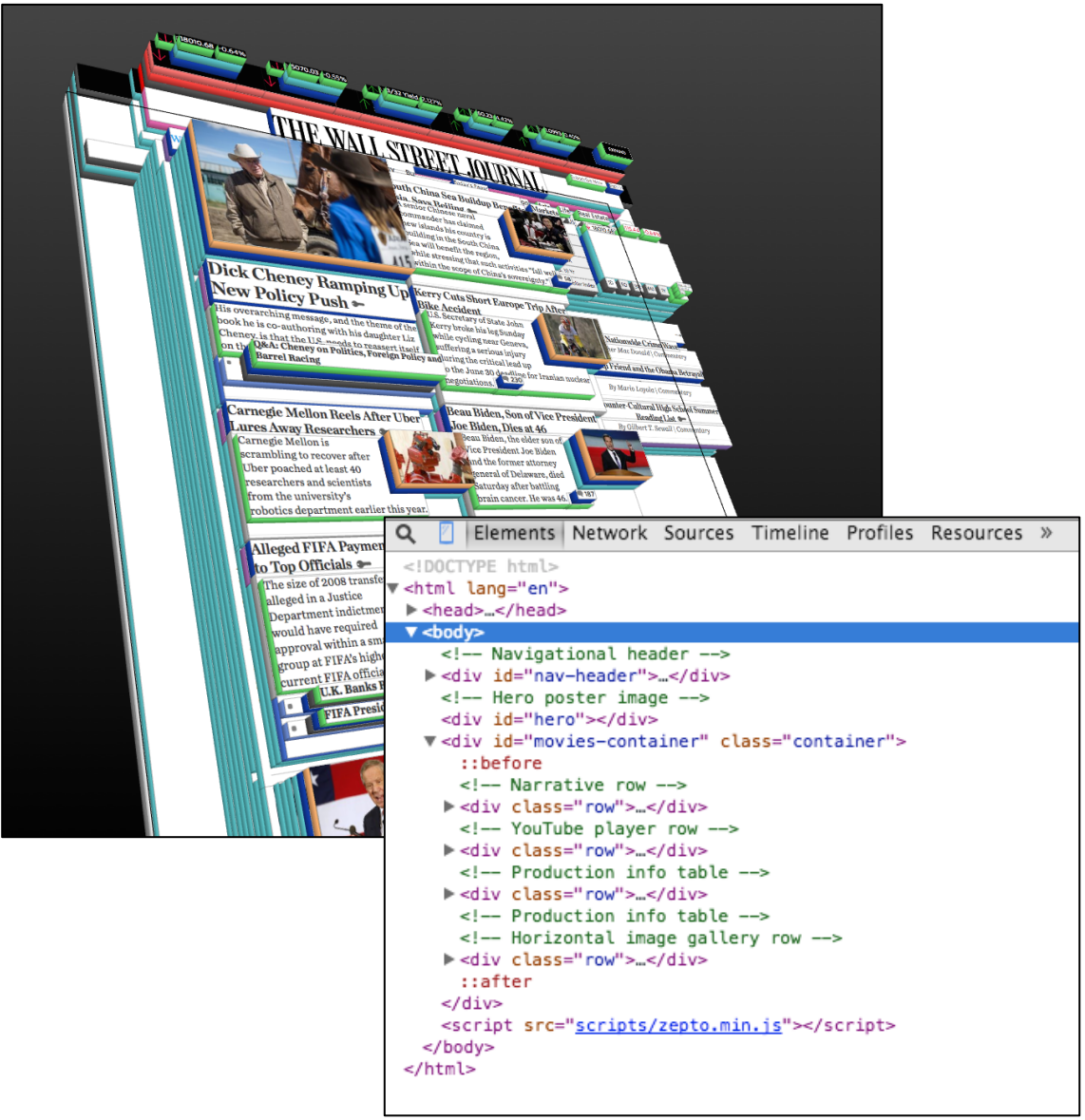
The DOM is the browser's in-memory representation of a web page. Think of it as holding the elements in your document (the stuff on your page).

The key purpose of the DOM is *programmatic* access to all the elements on your page: you can use JavaScript to add/remove/rearrange elements, style them with CSS, listen for events such as clicks and taps.

The key to great interactive experiences is JavaScript manipulation of the DOM, the focus of FEWD 32.

Principle

Browser Developer Tools



In all modern browsers there are developer tools that let you visualize what's going on "under the hood."

In Chrome you can right-click on any part of your page and **Inspect Element.** This will open up Developers Tools, highlighting the HTML element you clicked and show the matching CSS styles for you to browse. Another way to get there is `View Menu > Developer > Developer Tools`

Remember the user doesn't see the Developer Tools. It's meant for you, the web developer, to inspect the DOM, see matching styles, and later on, tinker with JavaScript.

Principle

HTML Elements

```
1  <div class="container"></div>
2  <div id="nav-header"></div>
3  <video id="my-video" src="goofy.mp4"></video>
4  <audio id="my-song" src="gymclass.mp3"></video>
```

HTML Elements encapsulate the content of your web page or web application. Some elements are very sophisticated, like `<video>` and `<audio>` elements capable of streaming rich content over the Internet. Others like the `<div>` are usually used to store and lay out text and images.

HTML elements capture the *what it is* rather than the *how it looks.* How it looks is controlled by CSS (see the slide *CSS Selectors*). Ideally the HTML element should not describe anything about how it looks, including styling, positioning, font, etc.

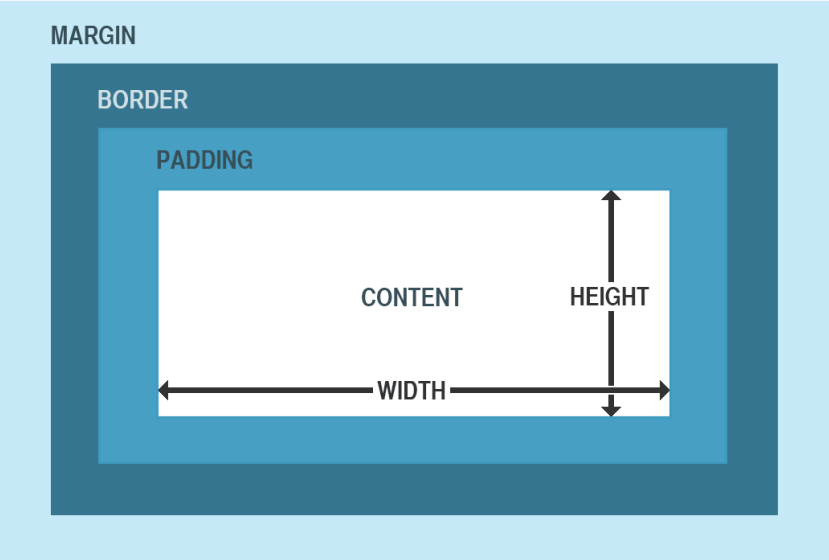
Separation of concerns is a common practice.

Elements have *attributes* like id, class, src, style, and the set of attributes is element-dependent (MDN is your friend). Just like the DOM, the key later on is using JavaScript to dynamically set an element's attributes. For example you can set the src of a video, an image, or audio element to change the media very easily.

By using JavaScript to programmatically set an element's CSS properties, you can create interactive, animated user interfaces.

Principle

The Box Model




```
#my-element {
  margin: 10px;
  padding: 25px;
  width: 100px;
  height: 100px;
  border: 1px solid black;
}
```

HTML elements -- be they div, span, a, video, you name it -- are governed by the CSS box model. At the center of the box is the **content** itself. Using CSS you can control the margin, border, padding, width, and height.

Like most things CSS, while it's good to be aware that the box model exists, to get good at using it requires repeatedly encountering it in the field: adjusting margins, getting widths to be just right, tweaking paddings.

Frameworks (patterns) like Twitter Bootstrap can help greatly with positioning elements responsively (e.g. in columns and stacking), but it's good to have this fine-grained control over an element's "bounding box" when needed.

Principle

CSS Selectors: id and class selectors

```html
<div class="container"></div>
<div id="nav-header"></div>
```

HTML file

```css
#nav-header {
  margin: 10px;
}

.container {
  padding: 25px;
}
```

CSS file

When writing HTML elements, you often want to style them using CSS. **IDs** and **classes** are two common ways to do this. In HTML, elements should have *unique* IDs while many elements can share the same class name.

Within CSS, you use **CSS selectors** to "match" your HTML elements. The two most common ones are the **ID selector** (#) and the **class selector** (.)

So given the CSS file to the left, the #nav-header ID selector will match the `<div id="nav-header"></div>` element in the HTML.

The .container class selector will match the `<div class="container"></div>` element in the HTML.

Other useful CSS selectors exist such as limiting the matching to only an element's children/descendants. At first I wouldn't worry too much about those and focus on ID (#) and class (.) selectors for your styling needs.

Principle

CSS Selectors: element selector

```
1  <div class="container"></div>
2  <div id="nav-header"></div>
```

HTML file

```
1  div {
2    background-color: black;
3  }
```

CSS file

Sometimes you want to match elements by *the tag itself.* That is, rather than try to match by an element's ID or an element's class, you might want to say "Give me all DIVs" or "Give me all SPANs" or "Give me all links (<a> tags)".

In the CSS file to the left, you're literally saying "for every single DIV on the page, make their background colors black."

Caveat: This is considered a match that's too broad. As your pages get bigger, you want to avoid matching across such a wide number of tags. This is why you should use id (#) and class (.) selectors whenever you can. They select such a small subset of all elements that it's usually efficient and fast for the browser to do. Contrast this to trying to match hundreds of DIVs or SPANs!

A common thing we'll see in later classes is "Give me all DIVs that are *children* of a particular element." So this also limits the matched set. I'll be sure to demonstrate this caveat in class, don't worry.

Principle

Color

```html
<div class="container"></div>
<div id="nav-header"></div>
```

HTML file

```css
.container {
  background-color: rgba(0, 0, 0, .5);
  color: gray;
}

#nav-header {
  color: #33F203;
}
```

CSS file

These are best learned by encountering them in the wild, but here's some helpful information about specifying color.

The CSS style `background-color` specifies just that: the color of the element's background. Just `color` on the other hand specifies the *foreground* color (usually text color).

You'll see on the left that there's 3 common ways to specify color:

- Using a reserved word like "gray" and "aliceblue" (really, there is a special color called aliceblue).
- A number in hex (hexadecimal) usually specified as 6 characters eg. #33F203. Designers can usually give these to you precisely.
- In terms of red, green, blue, alpha. You can arrive at unique colors using a combination of r, g, b. The alpha means opacity with 0 being totally transparent and 1 being totally opaque.
- Use RGBA when you want to have transparent backgrounds of your elements.

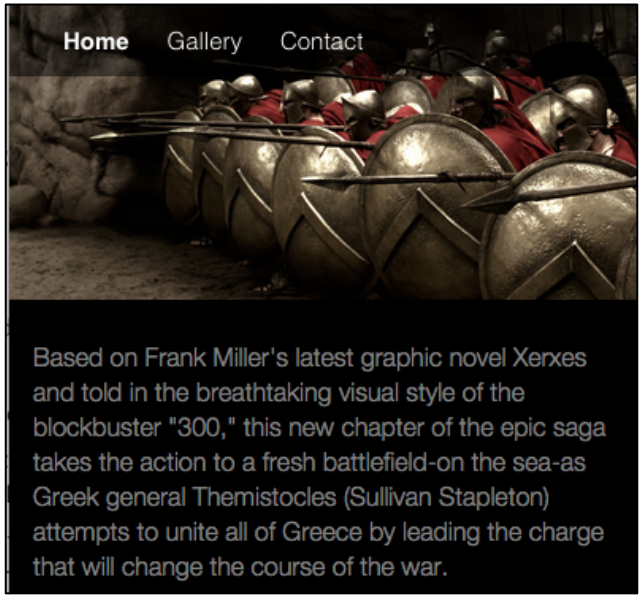
Principle

CSS display property: Inline vs. Block elements

When dealing with common tags like , and <a> it's helpful to know about inline vs. block elements.

You can force inline versus block using the CSS property **display**. Basically an element with CSS display set to **block** will cause a line break before and after the element (causing wrapping). The element will also naturally take up the full width available to it without you having to do anything.

An element with CSS display set to **inline** only takes up the space it needs (based on its content) and does *not* force line breaks/wrapping.



This is super confusing to try and work out in the abstract. For now it's enough to know that 's are display: block by default and 's and <a>'s are display: inline by default.

So in our 300 page, we use 's for the Home, Gallery, and Contact links so that they flow "inline" and DIVs for blocks of content that we want to wrap. Using just these simple combinations + Twitter Bootstrap you can lay out great pages / user interfaces.

Principle

CSS position property: static vs. absolute


```
/* Nav Header */
#nav-header {
  position: absolute;
  width: 100%;
  height: 47px;
  background-color: rgba(0, 0, 0, .5);
}
```

Take a look at the above (the DIV that holds the Home, Gallery, and Contact links). You'll notice that in the CSS file we set its position to **absolute** (fixed also works but ignore fixed for now).

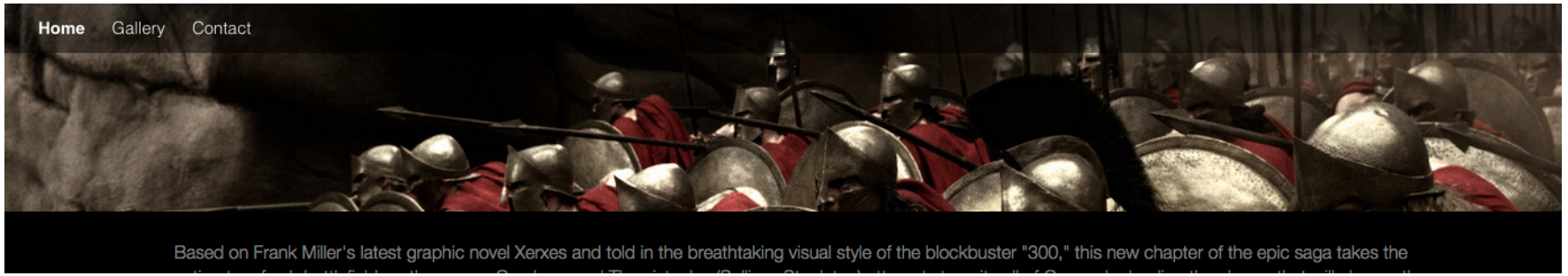
By default, the position is set to **static.** The browser will draw elements as it encounters them. If the position of an element is static -- the default -- that means the browser will draw the element in place where it naturally occurs. Neighboring elements will "scoot" because the **statically** positioned element takes up space in the document. As you'd expect.

Contrast this to setting the position to **absolute**. The browser does *not* reserve space in the document for absolutely positioned elements. Think of absolute position as **ripping the element out of the document** so that you can position it freely, and its neighbors will be none the wiser. That is, absolutely positioned elements are ripped out of the normal layout flow.

Whew, OK so what the heck does this mean? Well, we wanted our #nav-header to sit *on top* of the awesome hero picture of the 300 spartans. I wanted to use a transparent background color to create a nifty transparent overlay. So by setting the #nav-header to position: absolute, the hero image of the spartans SCOOTED UP as if the nav-header wasn't there. In fact, try changing the position from absolute (or fixed) **back to static.** See how the hero image scoots down now that the header isn't ripped out of the normal layout? In summary, use absolute when you want to freely position an element and cause its neighbors to ignore it. Useful for when you want to disrupt the normal layout flow of elements in a document.

Pattern

Navigational Header



In both web pages and mobile apps, it's common to have a horizontal header stretch across the top typically with navigational or filtering controls. Because it's fixed to the top and not expected to be responsive with the body content, this element can be placed outside the bootstrap typically shortly after your <body> tag.

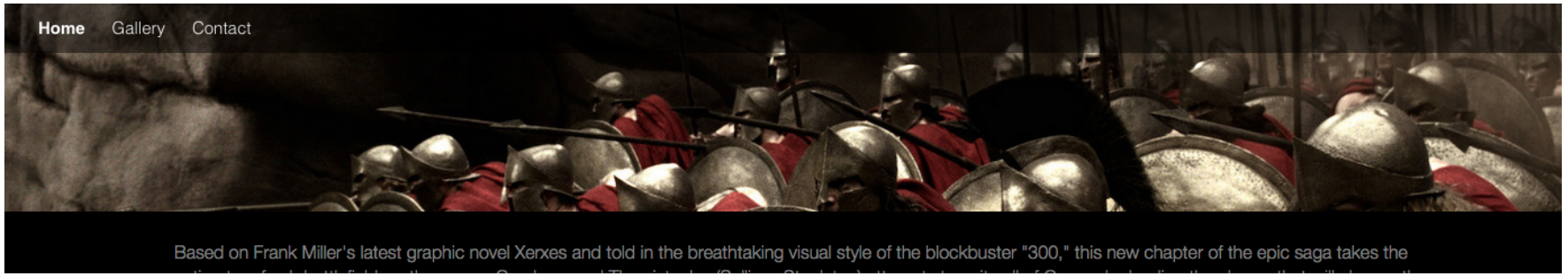
Here we use position: fixed to "fix" the bar at the top and set its width to 100%. The height is adjustable according to what will look good. We force the div to be "on top" of other content with a high z-index value, and achieve a transparent background with rgba (the latest parameter "a" stands for alpha transparency: 0 is invisible, 1 is opaque).

```
/* Nav Header */
#nav-header {
  position: fixed;
  top: 0;
  width: 100%;
  height: 47px;
  background-color: rgba(0, 0, 0, .5);
  z-index: 900;
}

/* All spans under #nav-header */
#nav-header > span {
  font-family: sans-serif;
  color: white;
  font-size: 15px;
  font-weight: 300;
  margin: 14px 10px;
  -webkit-font-smoothing: antialiased;
  display: inline-block;
}
```

```
<!-- Navigational header -->
<div id="nav-header">
  <span></span>
  <span><a class="active" href="index.html">Home</a></span>
  <span><a href="#">Gallery</a></span>
  <span><a href="#">Contact</a></span>
</div>
```

Pattern

Hero Image (using cover)




```
/* Hero poster */
#hero {
  background-image: url(300-hero.jpg);
  background-size: cover;
  height: 188px;
}
```

Modern browsers support background-size values of **cover** and **contain**, convenient for scaling images based on the size of their container. Rather than using an <img> element we use a for this purpose.

For a div whose ID is “hero” we set its CSS background-image to a url (could be a local file or any URL over the Internet). Because we want to scale the image to fit many viewport widths on mobile devices, we set background-size to *cover*. Experiment with cover vs. contain.

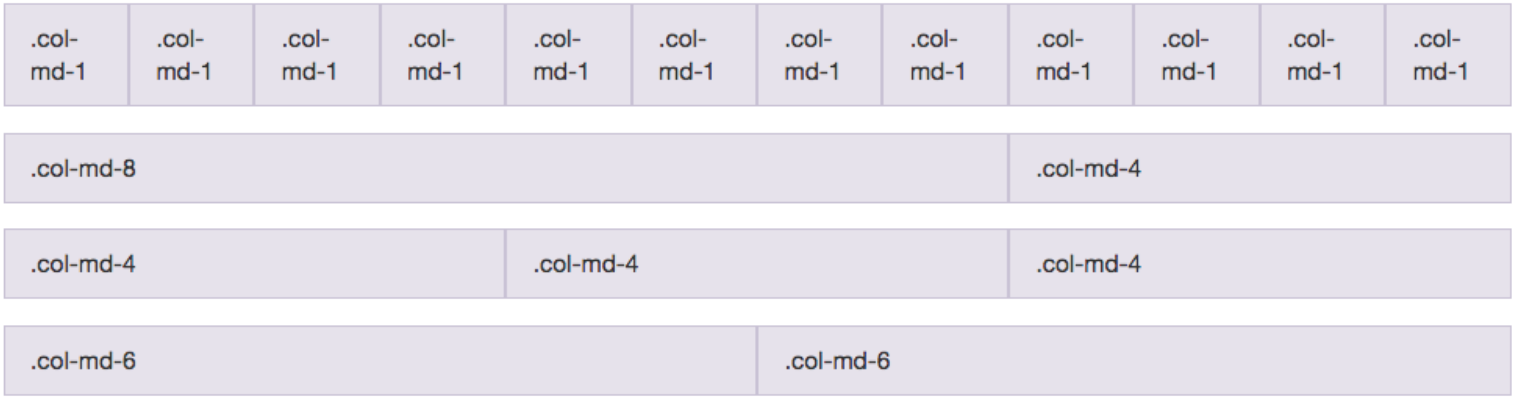
The background-size property has many nuances and tweaks you can study here, but for now here is the handy hero pattern. One thing to note is that your hero image should be pretty big if you want to scale all the way up to desktops/living room screens which can get costly fast. Be wary of serving gigantic images to mobile devices (use media queries).

Pattern

Bootstrap grid system

Bootstrap's guide to getting started is pretty good and friendly.

Twitter Bootstrap lets us lay out our web content against a 12-column grid. We can specify that content stretches across these columns, automatically stacking vertically when the viewport gets too small.



Bootstrap only asks that we do all of this within their Containers and Rows convention. So any content we want to “be responsive” -- that is, maintain a side-by-side relationship horizontally when the viewport is big enough, then automatically stack vertically when the viewport shrinks -- should get enclosed in:

```
<div class=”container”>
  <div class=”row”>
    (content goes here, other DIVs usually)
  </div>
</div>
```

Pattern

Bootstrap grid system (con't)

Your challenge is to assign classes to the content you want to flow responsively. Do this using one of Bootstrap's "grid classes" shown above. For example, this means "lay out the Dog, Cat, and Moose evenly with each taking up 4 out of the 12 columns."

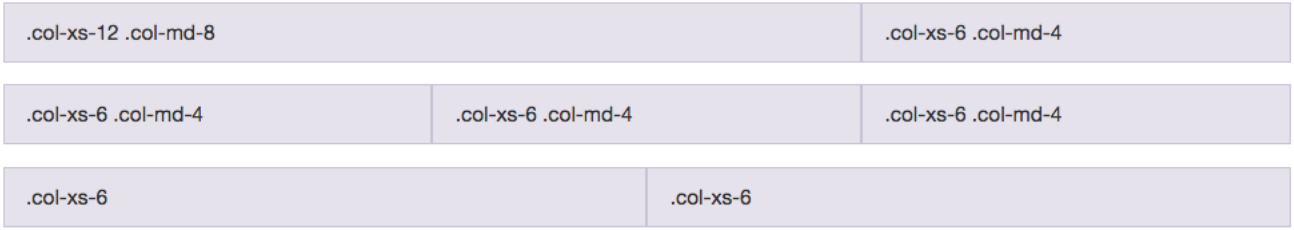
```
<div id=”container”>
  <div id=”row”>
    <div class="col-xs-4 panel">Dog</div>
    <div class="col-xs-4 panel">Cat</div>
    <div class="col-xs-4 panel">Moose</div>
  </div>
</div>
```

Responsively laying out the Dog, Cat, and Moose texts is pretty boring but remember you can put anything you desire in place of these texts: videos, IFrames (remember YouTube player?), images, entire swipe galleries, and of course other DIVs. In this way Bootstrap lets you responsively lay out generic content using its 12 column grid.

Pattern

Bootstrap grid system (con't)

	Extra small devices Phones (<768px)	Small devices Tablets (≥768px)	Medium devices Desktops (≥992px)	Large devices Desktops (≥1200px)
Grid behavior	Horizontal at all times	Collapsed to start, horizontal above breakpoints		
Container width	None (auto)	750px	970px	1170px
Class prefix	`.col-xs-`	`.col-sm-`	`.col-md-`	`.col-lg-`



Just like you can add multiple classes to your HTML elements, Bootstrap lets you add special classes corresponding to different sizes of viewports: from extra small devices, small devices, to medium then large and above.

Take a look at the first row of the purple table above: any **col-xs-*** class "kicks in" (applies) when the device viewport is extra small (phones). Any **col-md-*** class kicks in on medium devices.

Getting your layouts to look right isn't an exact science. The best way is to fire up your web page in your browser, then resize your browser to simulate all possible viewports. And of course, the best way to learn this is to try and lay out some real responsive web pages. None of this really sinks in in the abstract.

Pattern

YouTube IFrame “component”

In an ideal world, any component you desire would be as easy to use as `<video>` or `<audio>`. A `<youtube>` component makes a lot of sense (and exists, but it has dependencies on some special libraries).

For now, there is a technique that uses iframes. iframes let you “embed” another document / web page onto yours.

It’s not as clean as setting attributes on a <video> tag, but if you look at the text string you see parameters like controls=0, showinfo=0, and most importantly the `G3Rzy7YqUVU` id. This is the unique YouTube ID that you can grab from the URL in your browser’s address bar when you visit any YouTube video’s page.

```
<iframe id="ytplayer" type="text/html" src="http://www.youtube.
com/embed/G3Rzy7YqUVU?controls=0&showinfo=0&origin=http://example.
com" frameborder="0"></iframe>
```

Pattern

.spacer class

```
/* Apply this to all .row elements to create breathing room.*/
.spacer {
  margin-top: 25px;
}
```

This is just a simple trick that comes in handy often. As you're adding new rows in bootstrap like:

They will become bunched up and "touch" each other vertically. You can easily create spacing between rows by simply adding the spacer class that will force a margin-top of 25 pixels.

This also demonstrates how *multiple* classes can apply to a single element. Contrast this with **id** which is unique throughout the entire document.

This will become very clear as we lay out several pages together in class.

Pattern

Swipeable Image Gallery

Mobile devices are heavily optimized for scrolling content, both horizontally and vertically. Using this we can create an illusion of a swipeable gallery where neighboring images are out of sight until you scroll.

```css
/* Horizontal image gallery */
.gallery {
  margin: 25px auto;
  overflow-x: scroll;
  white-space: nowrap;
  -webkit-overflow-scrolling: touch;
}

.gallery img {
  width: 90%;
}
```

```html
<!-- Horizontal image gallery row. We'll discuss how
<div class="row">
  <div class="gallery">
      <img src="assets/gallery1.jpg">
      <img src="assets/gallery2.jpg">
      <img src="assets/gallery3.jpg">
      <img src="assets/gallery4.jpg">
      <img src="assets/gallery5.jpg">
      <img src="assets/gallery6.jpg">
  </div>
</div>
```

The key strategy is to shove a bunch of <img> elements inside our whose class we set to gallery. The key properties at work are `overflow-x: scroll` and `white-space: nowrap`.

`white-space: nowrap` suppresses wrapping to make it so that all the images flow side by side, just like text.

`overflow-x: scroll` means that if the content -- the images in this case -- overflows this element's left-right boundaries, allow the user to scroll to see the content that's cut off. If this was set to *hidden* it would just cut off the overflow and you couldn't scroll to see other images.

APPENDIX

Content

Week 1 (Web Applications Overview)

- Industry overview, trends, where to focus your skills to prepare for the next 5-10 years of computing (*Our industry does not respect tradition, it only respects innovation. -Satya Nadella*)
- Web application architecture (High-level overview *HTML/HTML5, CSS, JS*)
- Visualize the DOM, introduce common HTML elements
- The Box Model for HTML elements
- CSS basics: Styling HTML elements with CSS
- Static layout of responsive websites (responsive grid with Twitter Bootstrap)
- Twitter Bootstrap's responsive grid: rows, columns, phone/tablet/desktop layouts
- Push project to Git using the Git application

Week 2 (Static layout of responsive sites, review of key principles/patterns)

- Formal review of principles & patterns from week 1 (this deck)
- Responsive project #2 and #3: Responsive Email Template, Product Landing page. These projects reuse many of the techniques in the 300 project, and introduce a slightly more intricate page layout.

Content

Weeks 3, 4, 5 (Combination of formal introduction to programming, JavaScript, and JQuery)

- Formal introduction to programming fundamentals via JavaScript
  - Functions and scope
  - Variables and data types
  - Conditionals
  - Loops
  - Arrays and Objects
  - Practice iterating through data returned from popular APIs

- JQuery: What's it for?
  - Dot notation
  - Common selectors #.>
  - Key JQuery functions: DOM creation, attributes, append, remove, children, parent, css, addClass/removeClass
  - Event handling (callback functions, on, click, keypress, the event object)
  - Preview of AJAX, SPA (single-page app)

- Code organization, best practices in software engineering
- Using JavaScript/JQuery to interact with HTML DOM elements
- Using JavaScript/JQuery to set CSS styling
- Using JavaScript/JQuery to create, remove, and append DOM elements
- Series of exercises using JavaScript to manipulate DOM based on JSON blueprint
- CSS3: using JavaScript for interactive CSS-based animations

Week 6 (User input via Forms, the What & Why of APIs)

- Even more JavaScript -- a revisit and additional practice from Week 3
- Forms
  - Basic form validation & submission
  - Common form elements <form>, <button>, <select>, <input>
- Dynamic content presentation using APIs, AJAX, JSON
- Tons of practice working with JSON / JavaScript objects
- Practice hitting common APIs useful for students' final project: Flickr, Twitter, YouTube (both iFrame and native API), Maps, any remote API based on student interest

Week 7 (Information systems, dynamic content based on APIs + JSON)

- “Full stack” information system: wire up a backend using either Parse or Firebase
- Perform simple queries, process and visualize data
- HTTP GET and HTTP POST

Week 8

- Web components
- Follow-your-nose week: review any areas that are unclear, begin exploring possible final projects

Week 9

Project time + help from TAs & Instructor

Weeks 10

Project time + help from TAs & Instructor

Principle

Fonts

```
1  <div class="container"></div>
2  <div id="nav-header"></div>
```

HTML file

- TBD

CSS file