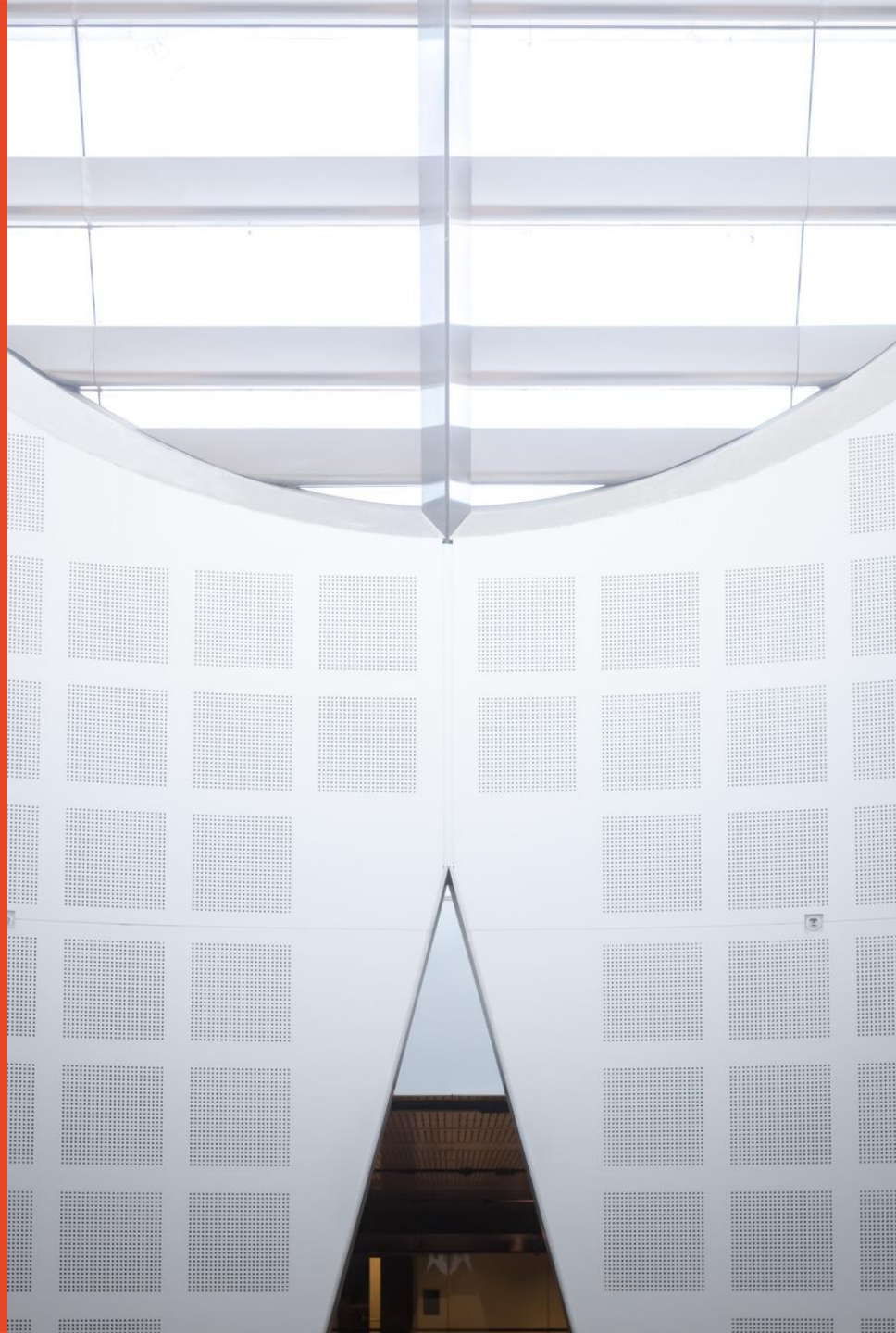
Web Application Development

COMP4347
COMP5347

The Browser and the Rendering Process

Week 4
Semester 1, 2025

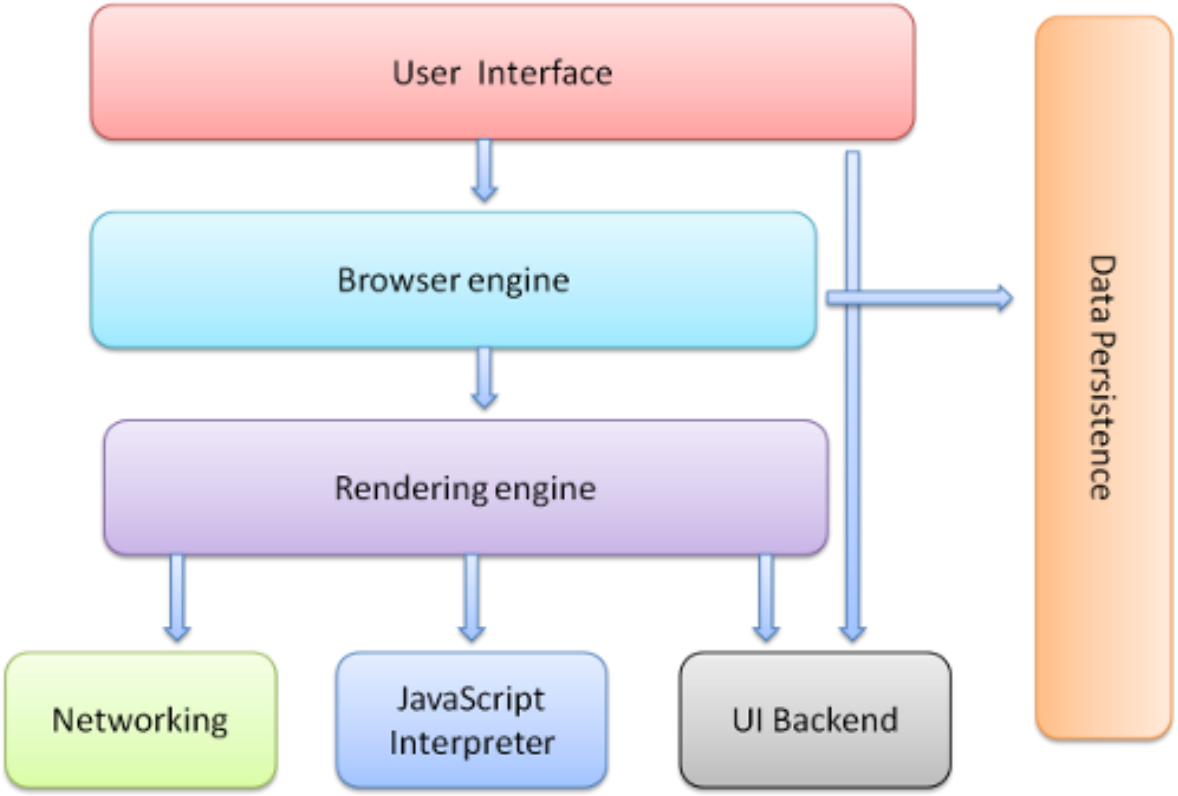
Dr. Mehdi Nobakht
School of Computer Science
THE UNIVERSITY OF SYDNEY



Outline

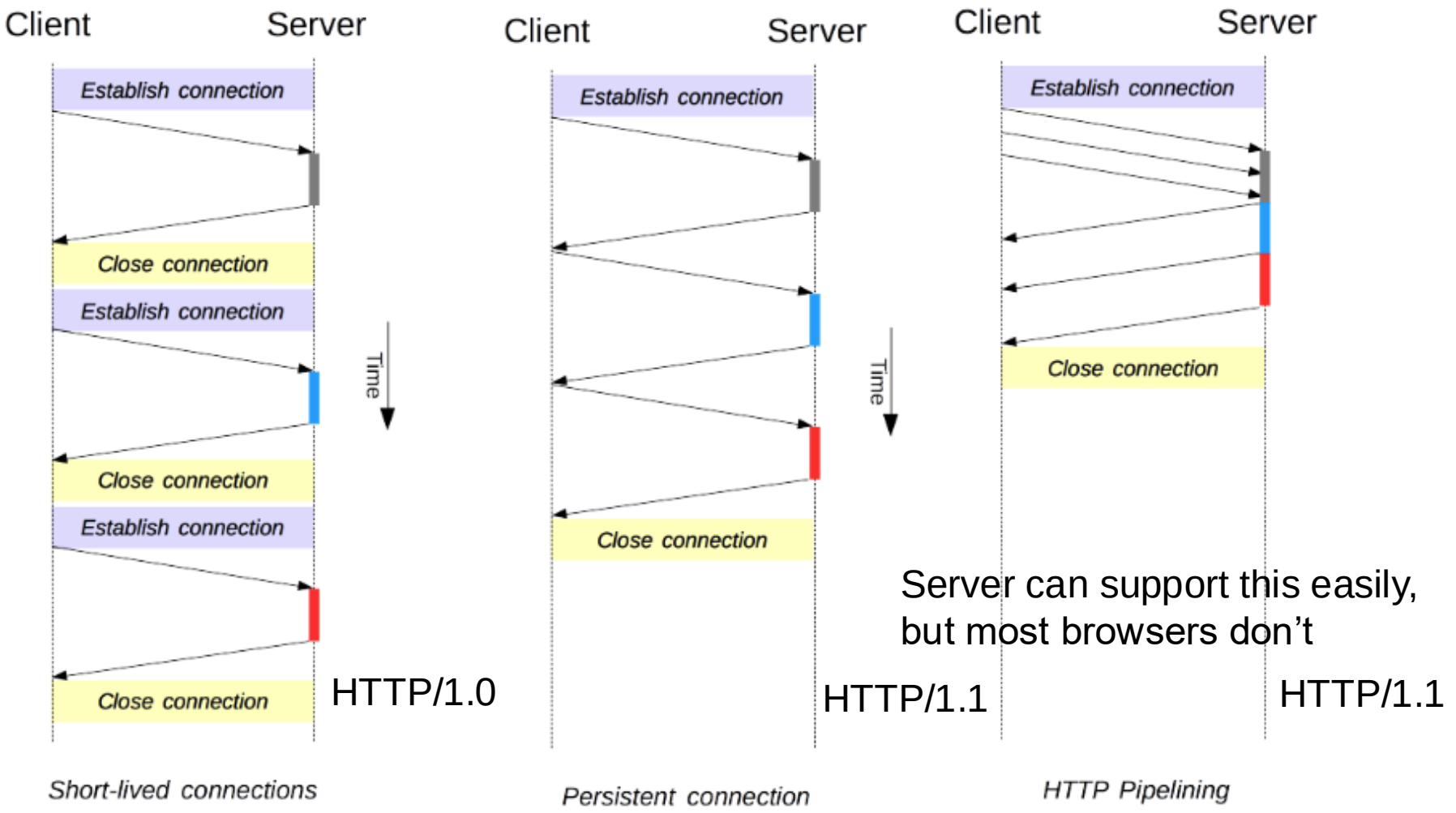
- Review of Browser
  - How browser works
  - HTTP Connection Management
  - HTTP Caching
- Browser Rendering Process
  - Critical Rendering Path
  - DOM and CSSOM
  - Render Path Analysis

How Browsers Work?



http://taligarsiel.com/Projects/howbrowserswork1.htm

HTTP Connection Management



https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x

HTTP Connections

- Short-lived
  - Own connection, sequential requests
  - Default in HTTP1.0 (in http1.1 "C*onnection*" header sent with "close")
- Persistent (keep-alive)
  - Reuse connection to send multiple requests
  - Idle connections will be closed ("*keep-alive*" header = min time open)
- Pipelining
  - Successive requests without waiting for a response

http://sgdev-blog.blogspot.com.au/2014/01/maximum-concurrent-connection-to-same.html

Parallel Connections

- All HTTP/1.x connection is serializing requests (without pipelining)

- To improve performance, browser open several connections to each domain, sending parallel requests
  - Max. no. of parallel connection small (not DoS attack)
  - Max. concurrent connections:
    - Chrome (4-23): 6 connections, Chrome (34): 8 connections
    - IE (8-9): 6 connections, IE(10): 8 connections
    - Safari (3-4): 4 connections
    - Firefox (4-17): 6 connections

http://sgdev-blog.blogspot.com.au/2014/01/maximum-concurrent-connection-to-same.html

Caching in HTTP

- Goal of caching in HTTP
  - Sending less requests
    - Less round-trips
    - “Expiration" mechanism
  - Sending less full responses
    - Less network bandwidth
    - “Validation" mechanism
- Level of caches
  - Server-side
  - Client-side (proxy and **browser**)
- Cache correctness
  - Response correctly served from the cache
  - Otherwise, communication error or warning

Caching in HTTP (cont)

- Expiration Model
  - Server-Specified Expiration
    - e.g., Cache-Control: no-cache
    - Cache-Control: max-age=60
  - Heuristic Expiration
    - Assign expiration times using heuristic algorithms

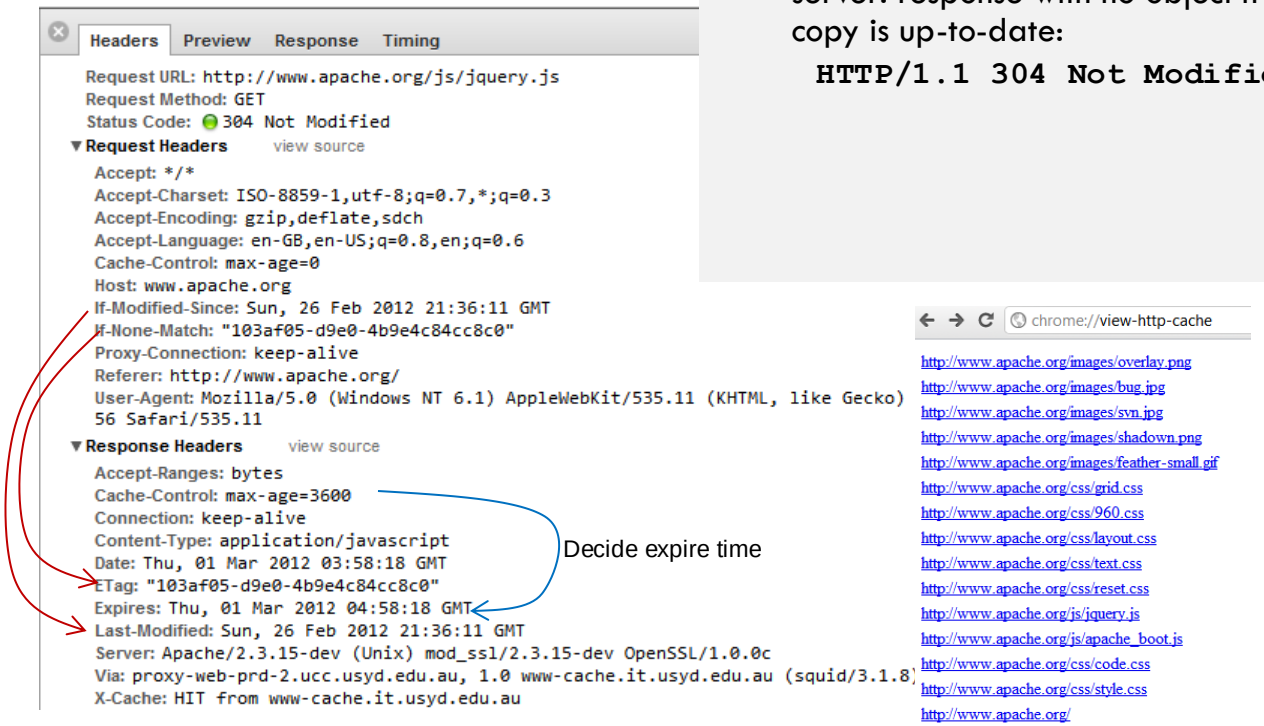
- Validation Model
  - When a cache has a stale entry that it would like to use as a response to a client's request, it checks with the origin server to see if its cached entry is still usable
  - Entry is still valid, no overhead of re-transmitting the whole response

Caching in HTTP (cont)

- Validation Model
  - Last-Modified Dates
  - Entity Tag Cache Validators
    - Entity tags are used for comparing two or more entities from the same requested resource.
  - Requestor side:
    - If-Match
      - If-Match: "xyzzy"
    - If-None-Match
    - If-Modified-Since
      - If-Modified-Since: Sat, 29 Oct 2018 19:43:31 GMT

Conditional GET: Browser caching

- **Goal:** don't send object if up-to-date cached
- client: date of cached copy in HTTP request
  `If-modified-since: <date>`
  `If-None-Match: <Etag>`
- server: response with no object if cached copy is up-to-date:
  `HTTP/1.1 304 Not Modified`



Outline

- Review of Browser
  - How browser works
  - HTTP Connection Management
  - HTTP Caching

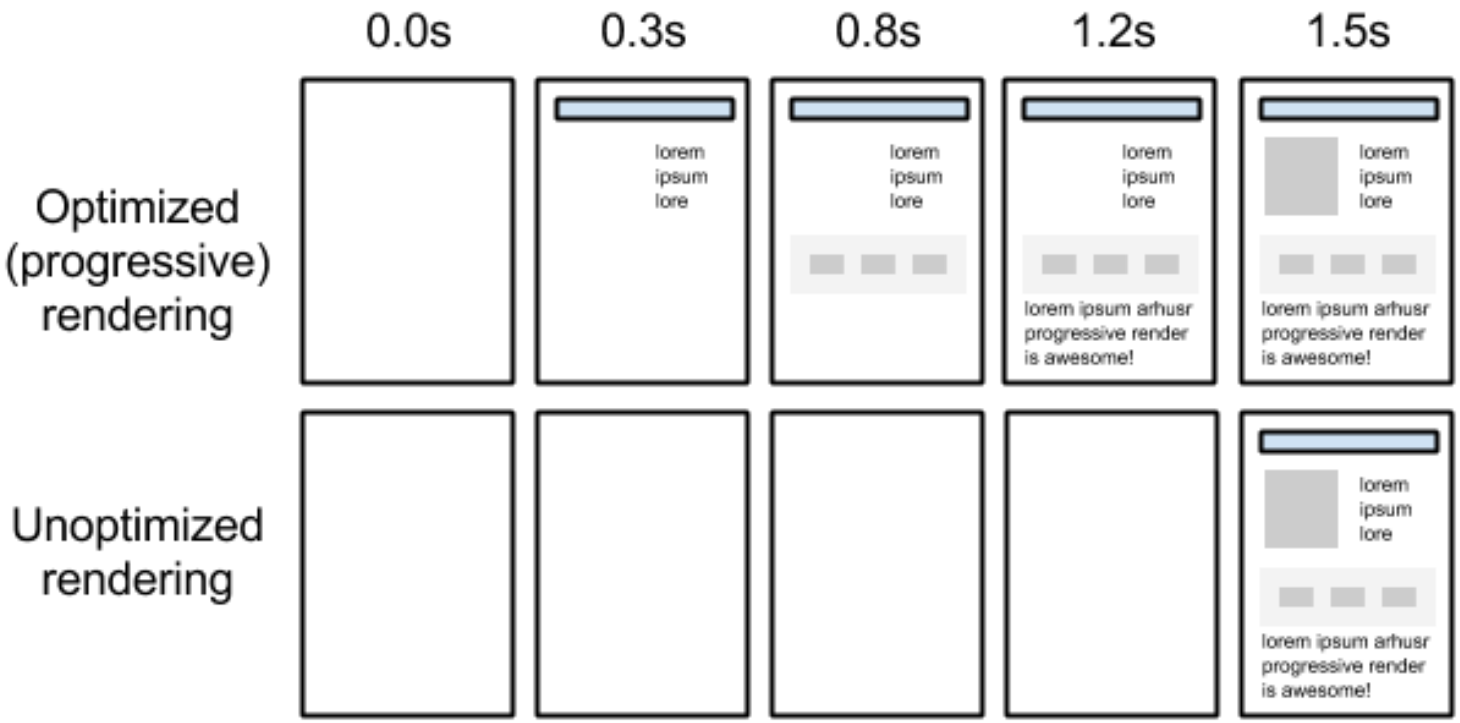
- **Browser Rendering Process**
  - **Critical Rendering Path**
  - **DOM and CSSOM**
  - **Render Path Analysis**

Critical Rendering Path

- Webpages to provide good user experience
  - Performance (speed), secure, intuitive design, …
- Think of user experience when there's long delay or non-responding pages
- Web developers write HTML, CSS and JavaScript (JS) and the browser displays content accordingly
- How the browser render webpages from corresponding HTML, CSS and JS?
- The actual steps browsers take to receive/parse/display data from web server is called critical rendering path
- Understanding the rendering process is important for performance optimization

Critical Rendering Path

– *Optimized (progressive) rendering* prioritize the display of a webpage content to minimize the total amount of time required to display the content



Overall Rendering Process

1. Process HTML elements and build the *DOM tree*

2. Process CSS rules and build the *CSSOM tree*

3. Combine the DOM and CSSOM into a *render tree*

4. Run layout on the render tree to compute geometry of each node

5. Paint them on the screen

Constructing the Object Model

- Document Object Model (DOM) for HTML
  - Each element inside a HTML document is represented as a node
  - Attributes and text between a pair of tags are also nodes
  - Nested element becomes the child node of its parent node
  - The whole HTML document can be represented as a tree called DOM tree

- Constructing the object model

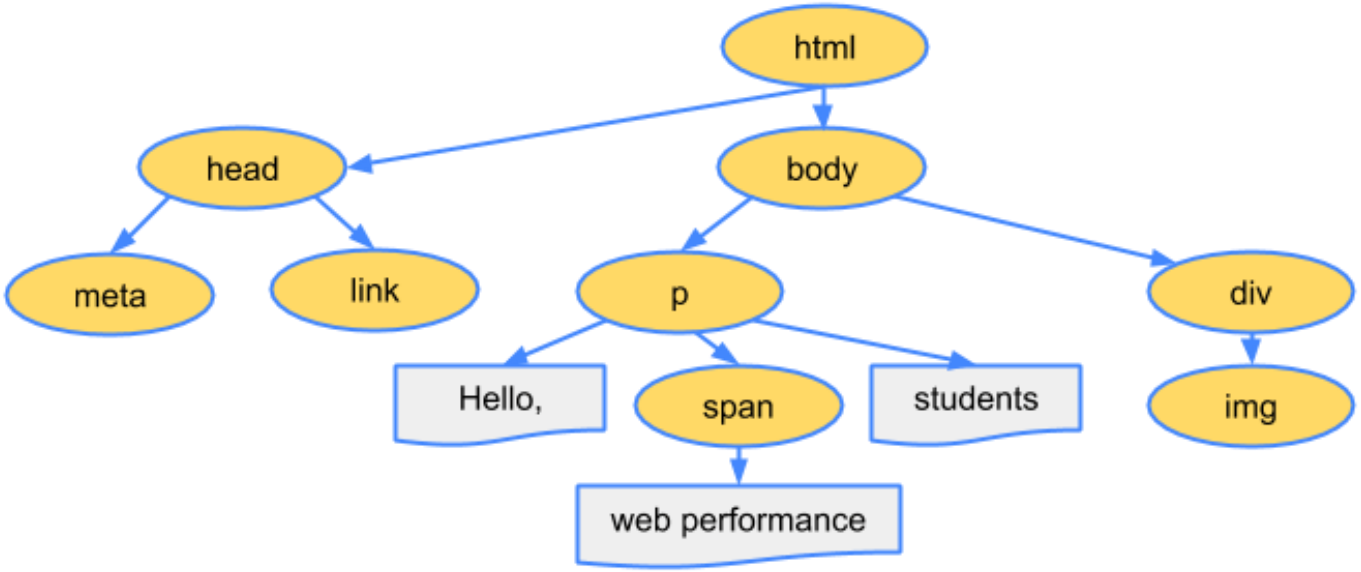
  Bytes → characters → tokens → nodes → object model

Constructing the DOM

- convert bytes to characters
- Converts strings into distinct tokens (W3C HTML5 standard)
- Convert tokens into objects (properties and rules)
- Construct a tree structure from the created objects

DOM Construction – Example

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```



Request Supporting Objects

- Requesting support objects happens at the same time while the DOM is constructed
  - E.g., `<link>` to CSS file → constructs the node → send a request to obtain the object specified by the link
  - E.g., `<img>` tag → constructs the node → sends a request to download the image

- After receiving the style sheet file, the browser starts to parse it and build a CSSOM tree

CSS Object Model (CSSOM)

- CSSOM defines generic parsing and serialization rules for CSS files, media queries and selectors
- A W3C standard with a working draft as of 17 Mar. 2016 (https://www.w3.org/TR/cssom-1/)

Constructing CSSOM

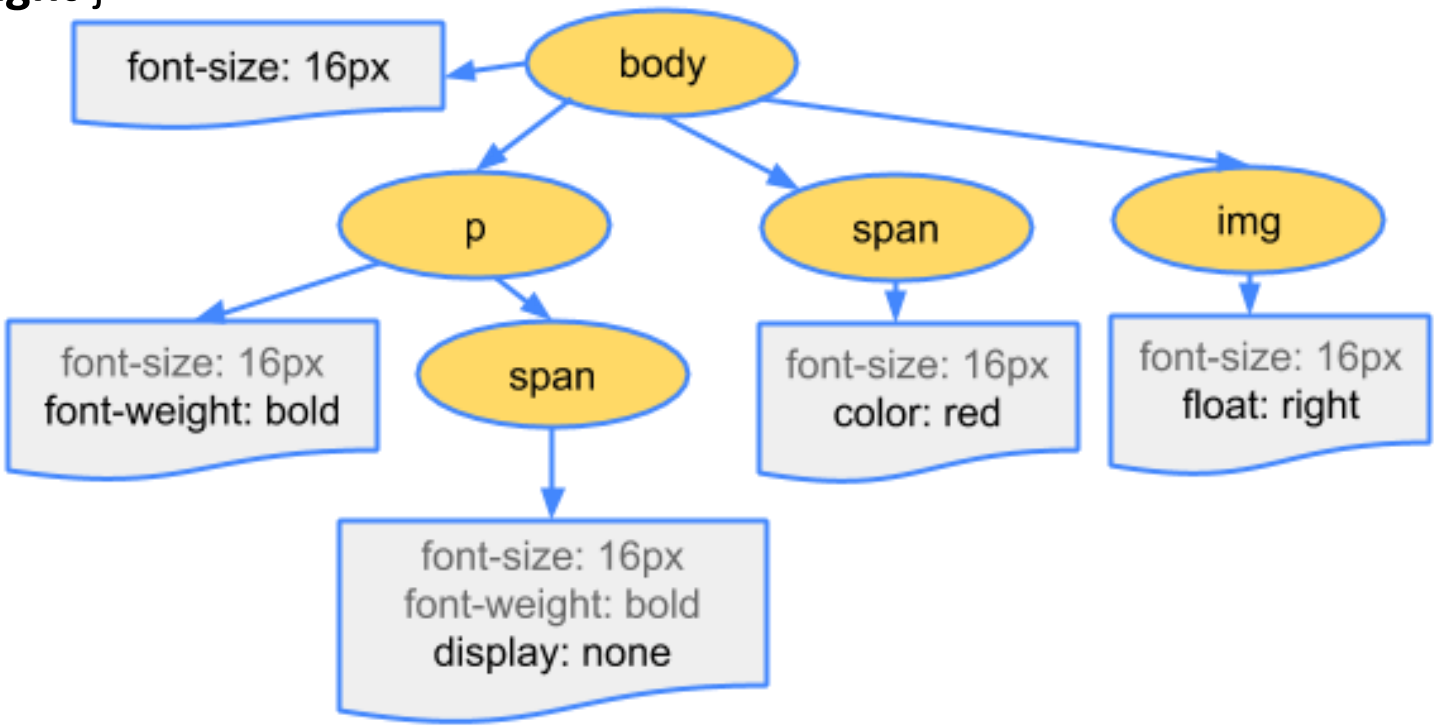
- DOM construction process but using CSSOM rules to construct the corresponding CSSOM

CSS Object Model (CSSOM)

- The tree structure is organized following the "*cascading*" principles
- The final set of rules an element has is the result of inheritance and conflict resolving
- Default browser's styles not shown, only *user agent styles* (overrides the browser defaults)
- Use tools to see the timeline of CSSOM

CSSOM – Example

```
body { font-size: 16px }
p { font-weight: bold }
span { color: red }
p span { display: none }
img { float: right }
```

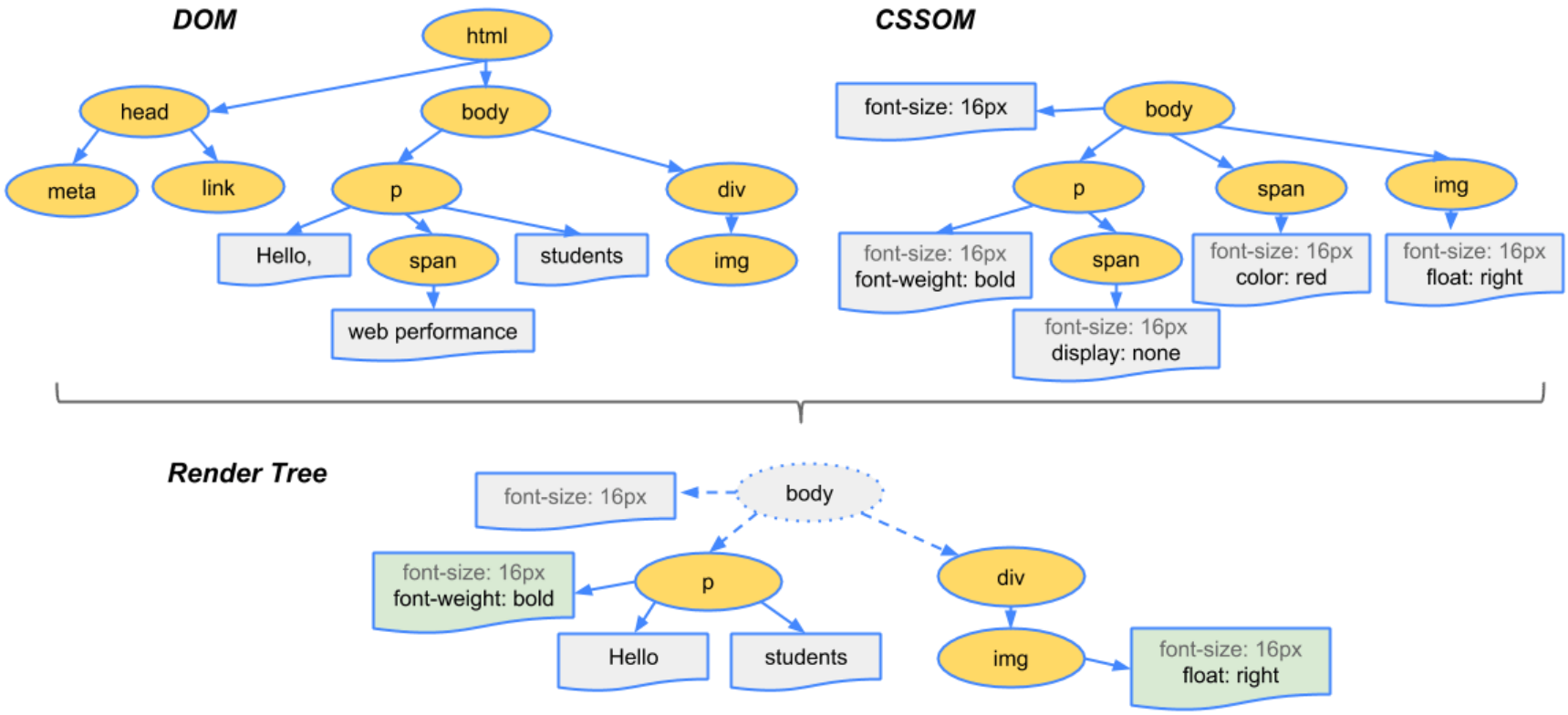


Render Tree Construction

- DOM and CSSOM objects are independent
- By merging both DOM and CSSOM
  - Contains only the nodes required to render the page

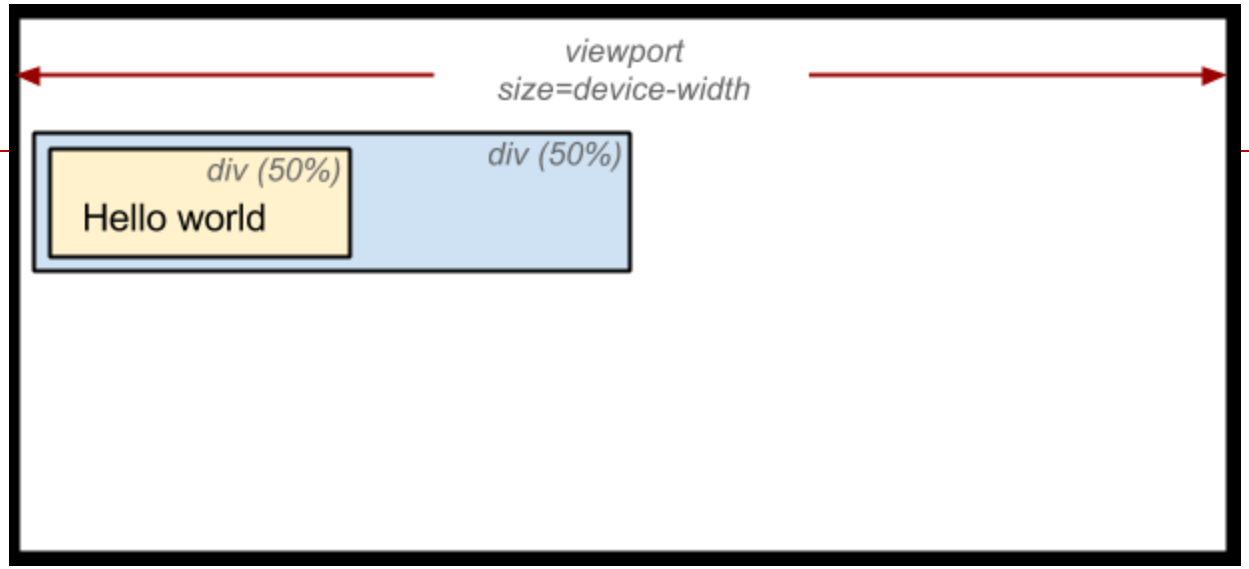
1. Traverse each visible node starting from the root of DOM tree
2. Find appropriate CSSOM rules for each visible node and apply it
3. Produce visible nodes with content and their computed styles

Render Tree Construction – Example



Layout (Reflow)

```
<html>
    <head>
        <meta name="viewport" content="width=device-width,initial-scale=1">
        <title>Critial Path: Hello world!</title>
    </head>
    <body>
        <div style="width: 50%">
            <div style="width: 50%">Hello world!</div>
        </div>
    </body>
</html>
```



Layout (Reflow)

- Computes the exact position and size of each object within the viewport of the device
- The output of the layout is a "box model" which captures the exact position of each element
- Last is the paint process which renders the pixels on the screen taking the final render tree
- Time to construct the render tree, layout and paint depends on the size of the document, styles used and the device it is running on

Render Blocking CSS

- HTML and CSS are render blocking resources
  - Browser needs to have all of them before it can start to display something
    - CSS links near the top of HTML page browser to obtain them early
  - CSS *media type* and *queries* to specifying some resources non-render blocking
  - Pages with multiple CSS to be used under different conditions
    - To print an article/email, all side bars should not appear
    - Screen size is too small, less important content can be hidden
  - CSS not intended for the current condition will not block the rendering process

Render Blocking CSS

– Classify each of the following CSS resources in terms of render blocking when the page is first loaded? Explain your answer

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

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

3. `<link href="portrait.css" rel="stylesheet" media="orientation:portrait">`

4. `<link href="print.css" rel="stylesheet" media="print">`

Render Blocking CSS

- <link href="style.css" rel="stylesheet">
  - Default media type is set to “all” if not specified.
- <link href="style.css" rel="stylesheet" media="all">
  - Render-blocking (applies to all media types, the default is “all”)
- <link href="portrait.css" rel="stylesheet“ media="orientation:portrait">
  - Could be both (evaluated when the page is loaded and depends on the device position when the page is loaded)
- <link href="print.css" rel="stylesheet" media="print">
  - Non-render blocking (only when the page is being in “print” mode)

JavaScript and DOM

- JS allows to modify every aspect of a page

- JS may block DOM construction and delay when the rendering
  - Depends on location of JS code
  - *Embedded* or *inline* script → may block DOM construction
  - Script tag → DOM construction pauses until the script finishes executing
  - External JS → stop constructing the DOM tree and wait for the file to be downloaded and executed → continue constructing the DOM tree

Embedded JavaScript Example

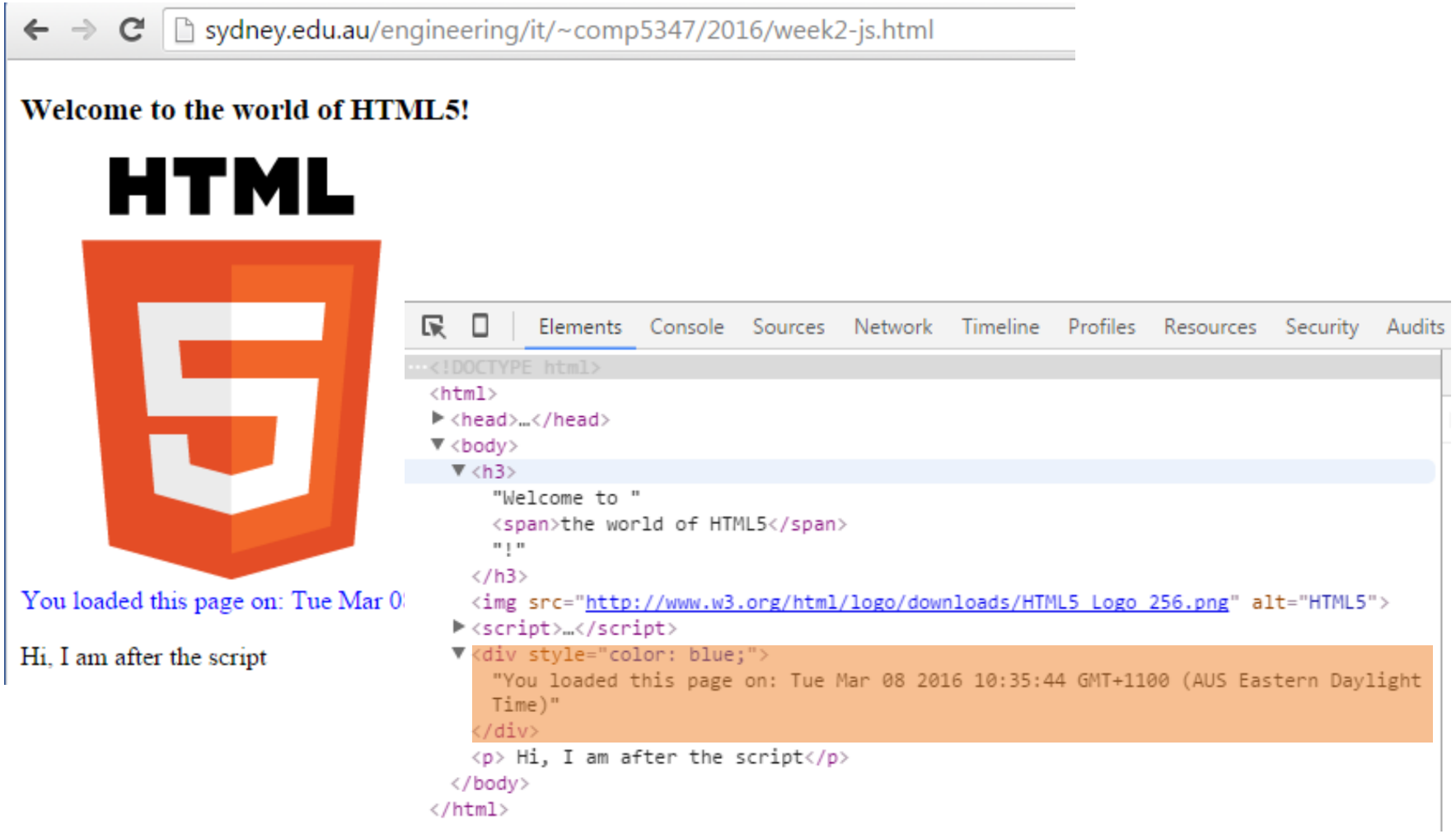
```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Week 2</title>
</head>
<body>

<h3>Welcome to <span>HTML5</span>!</h3>
<img src="http://www.w3.org/html/logo/downloads/HTML5_Logo_256.png" alt="HTML5">
<script>
    var span = document.getElementsByTagName('span')[0];
    span.textContent = 'the world of HTML5'; // change DOM text content
    // create a new element, style it, and append it to the DOM
    var loadTime = document.createElement('div');
    loadTime.textContent = 'You loaded this page on: ' + new Date();
    loadTime.style.color = 'blue';
    document.body.appendChild(loadTime);
</script>
<p> Hi, I am after the script </p>
</body>

</html>
```

document object represents the current page, it is the starting point to access all other HTML elements

individual element's style is accessed using this syntax: ***element*.style.*property***

Document.body is a convenient way of returning the body element

Output of the Embedded JS xample



Global Variable Example

```
<html>
<head>
<meta charset="UTF-8">
<title>Week 2</title>
</head>
<body>
<h3>Welcome to <span>HTML5</span>!</h3>
<img src="http://www.w3.org/html/logo/downloads/HTML5_Logo_256.png" alt="HTML5">
<script>
      var span = document.getElementsByTagName('span')[0];
      span.textContent = 'the world of HTML5'; // change DOM text content
      // create a new element, style it, and append it to the DOM
      var loadTime = document.createElement('div');
      loadTime.textContent = 'You loaded this page on: ' + new Date();
      loadTime.style.color = 'blue';
      document.body.appendChild(loadTime);
</script>
<p> Hi, I am after the script </p>
<script>
      var anotherLoadTime = document.createElement('div');
      anotherLoadTime.innerHTML = loadTime.innerHTML
      document.body.appendChild(anotherLoadTime);
</script>
…..
```

Global Variables Example (cont'd)

Welcome to the world of HTML5!

You loaded this page on: Mon Mar 27 2017 16:38:18 GMT+1100 (AUS Eastern Daylight Time)

Hi, I am after the script

You loaded this page on: Mon Mar 27 2017 16:38:18 GMT+1100 (AUS Eastern Daylight Time)

Asynchronous JavaScript

- By default all JS is parser-blocking

- The browser's behaviour and allow it to continue to construct the DOM and let the script execute when it is ready
  - Mark the script using the "*async*" keyword

```
<script src="app.js" async>
</script>
```

Outline

- Review of Browser
  - How browser works
  - HTTP Connection Management
  - HTTP Caching

- **Browser Rendering Process**
  - **Critical Rendering Path**
  - **DOM and CSSOM**
  - **Render Path Analysis**

Measuring Critical Rendering Path (CRP)

- Performance optimization requires good measurement and instrumentation approach
- "*You cannot optimize what you cannot measure*"
- One approach for measuring CRP is the Navigation Timing approach using an API

https://www.w3.org/TR/navigation-timing-2/

Navigation Timing API

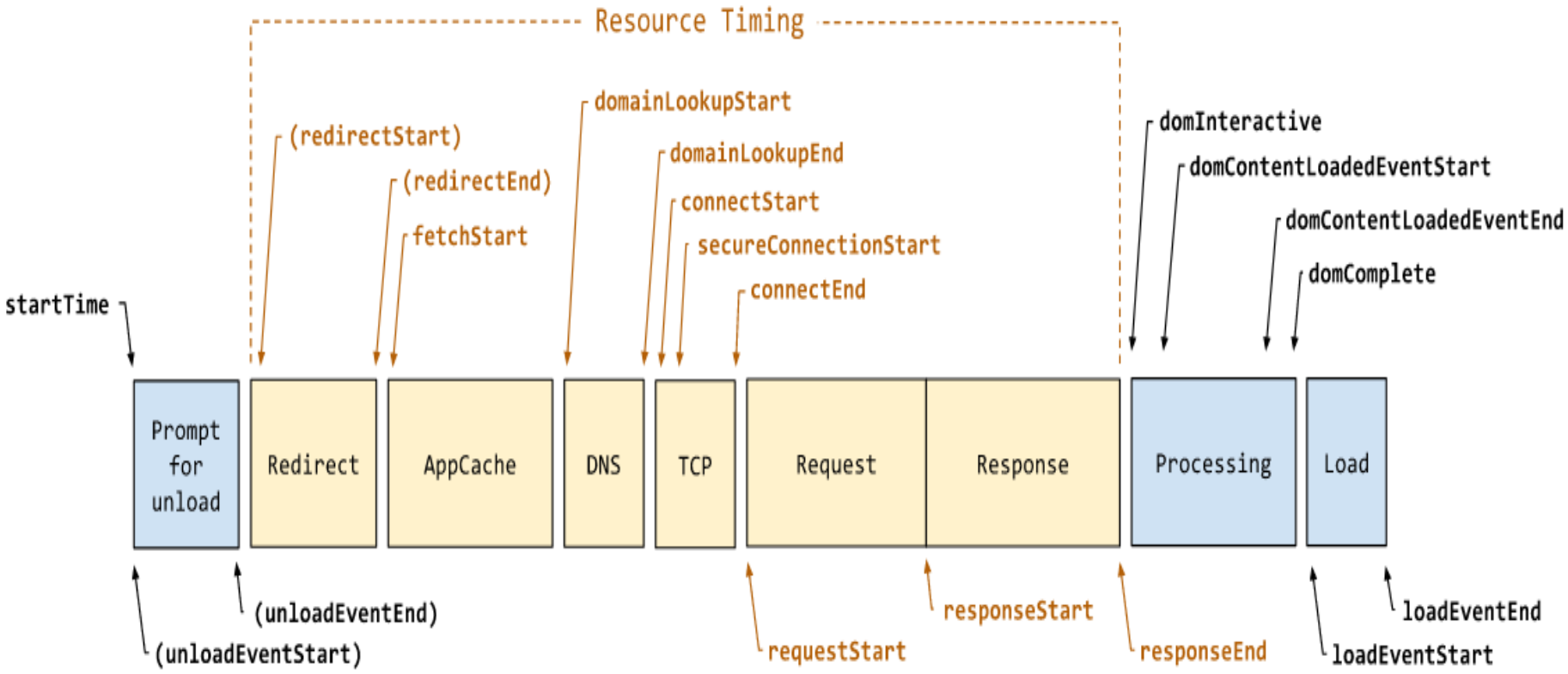
- An interface for web application to access the complete timing information for navigation of a document

- Another W3C working draft
  - Browsers are expected to implement to capture the time of various stage and also to fire relevant events
  - JavaScript codes are able to access the timing information and to listen to the event

- "Navigation started by clicking on a link, or entering the URL in the user agent's address bar, or form submission, or initializing through a script operation other than the ones used by reload and back/forward".

https://www.w3.org/TR/navigation-timing-2/

Overall Navigation Timing 2 Process



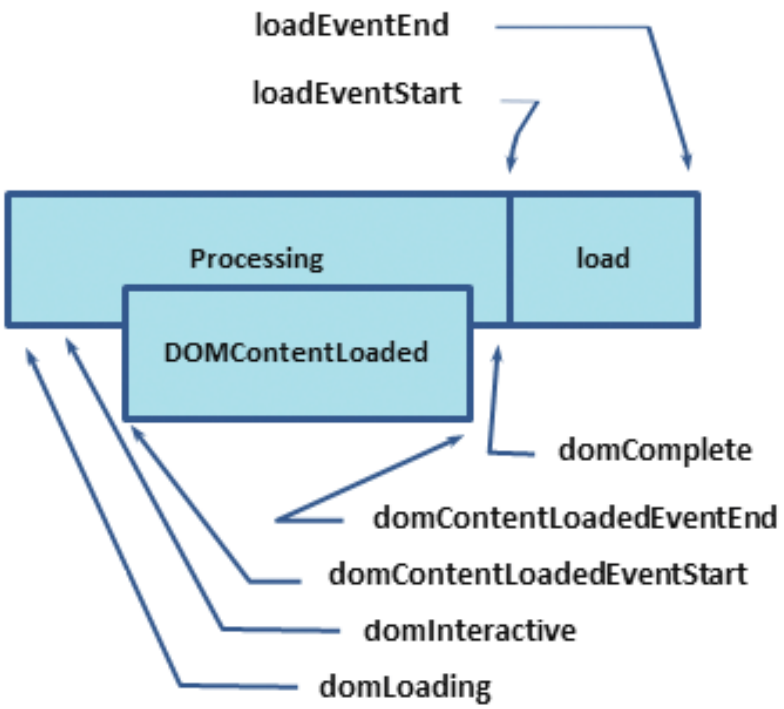
https://www.w3.org/TR/navigation-timing-2/

Process related with Rendering

domLoading: the starting timestamp of the entire process

domContentLoaded: when browser has finished parsing the HTML document the DOM is constructed.

domComplete: all of the processing is complete and all of the resources on the page (images, etc.) have finished downloading, **onLoad** event will file

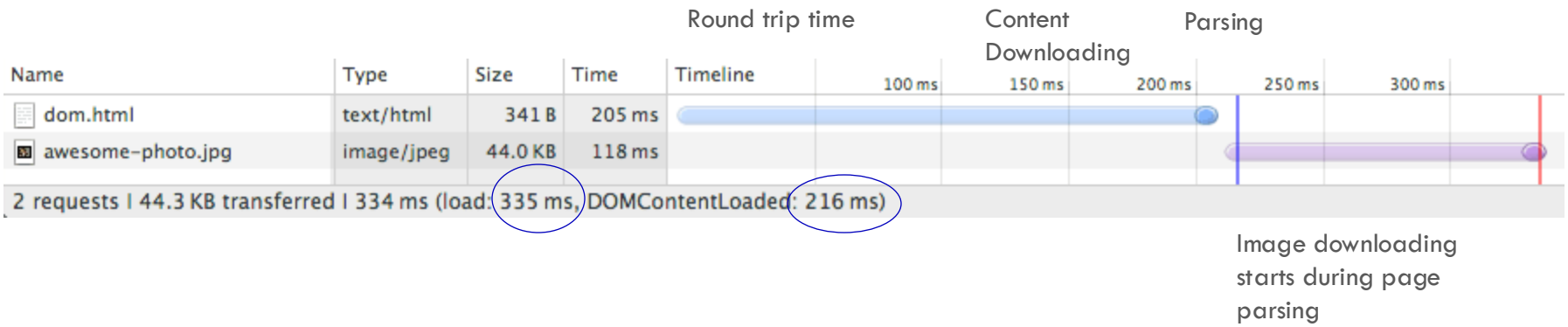


Analyzing Critical Rendering Path Performance

- Simple page with only HTML and an image
- A more complex page with HTML, an image and external CSS and JS file
- An example with HTML, an image and embedded CSS and JS file

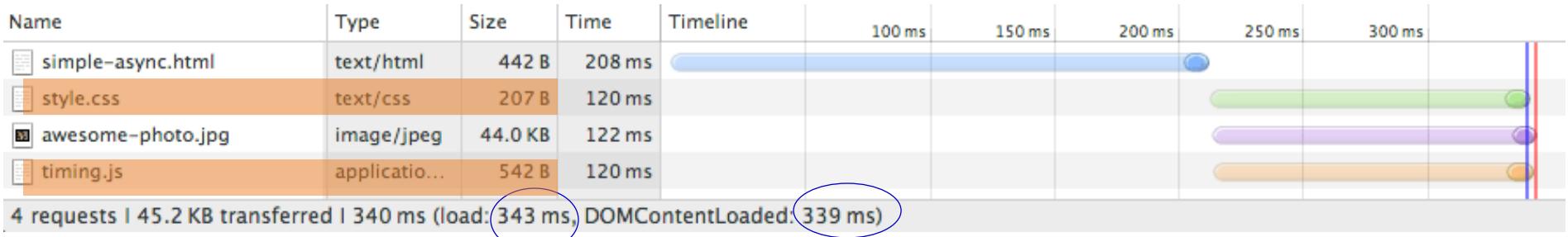
A page with HTML and image

```
<html>
 <head>
<meta name="viewport" content="width=device-width,initial-scale=1">
 <title>Critical Path: No Style</title>
</head>
<body>
<p>Hello <span>web performance</span> students!</p>
<div><img src="awesome-photo.jpg"></div>
</body>
</html>
```



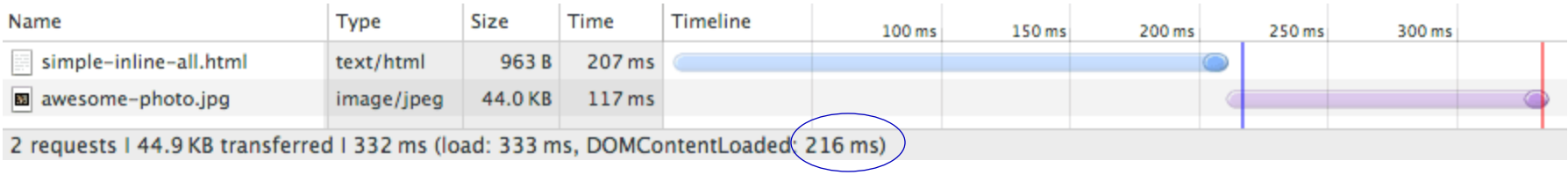
A Page with external CSS and JS file

```
<html>
<head>
<title>Critical Path: Measure Script</title>
 <meta name="viewport" content="width=device-width,initial-scale=1">
<link href="style.css" rel="stylesheet">
</head>
<body onload="measureCRP()">
<p>Hello <span>web performance</span> students!</p>
 <div><img src="awesome-photo.jpg"></div>
<script src="timing.js"></script>
 </body>
</html>
```



Page with embedded CSS and JS

```
<html>
 <head>
<title>Critical Path: Measure Inlined</title>
<meta name="viewport" content="width=device-width,initial-scale=1">
 <style>  p { font-weight: bold } .....   </style>
</head>
 <body>
   <p>Hello <span>web performance</span> students!</p> <div>
   <img src="awesome-photo.jpg"></div>
   <script>
      var span = document.getElementsByTagName('span')[0];
     ...
    </script>
</body>
</html>
```

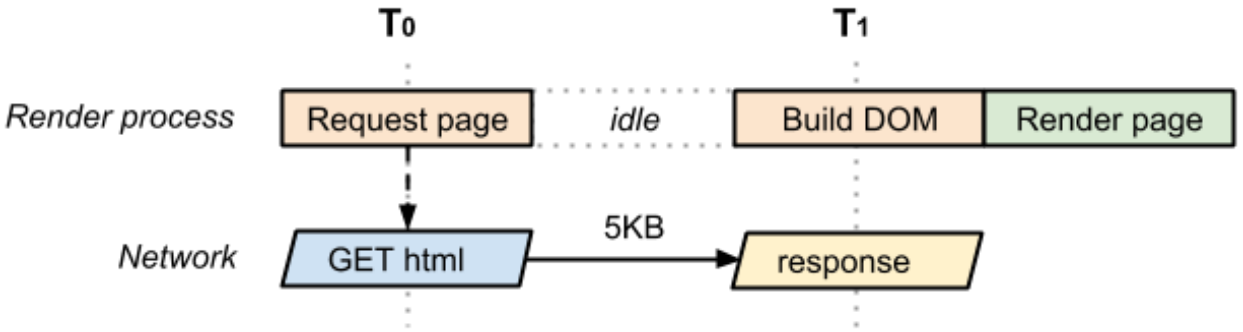


Performance Patterns

- **Critical Resource**: resource that needs to be downloaded before rendering the page
  - HTML, CSS, and JavaScript
- **Critical Path Length**: number of round trips to fetch all critical resources; ignore the initial tcp connection set up time
- **Critical Bytes**: total amount of bytes required get before rendering the page (sum of the transfer file sizes of all critical resources)

Page with only HTML and image

```
<html>
<head>
<title>Critical Path: Measure Script</title>
 <meta name="viewport" content="width=device-width,initial-scale=1">
</head>
<body>
<p>Hello <span>web performance</span> students!</p>
 <div><img src="awesome-photo.jpg"></div>
</body>
</html>
```

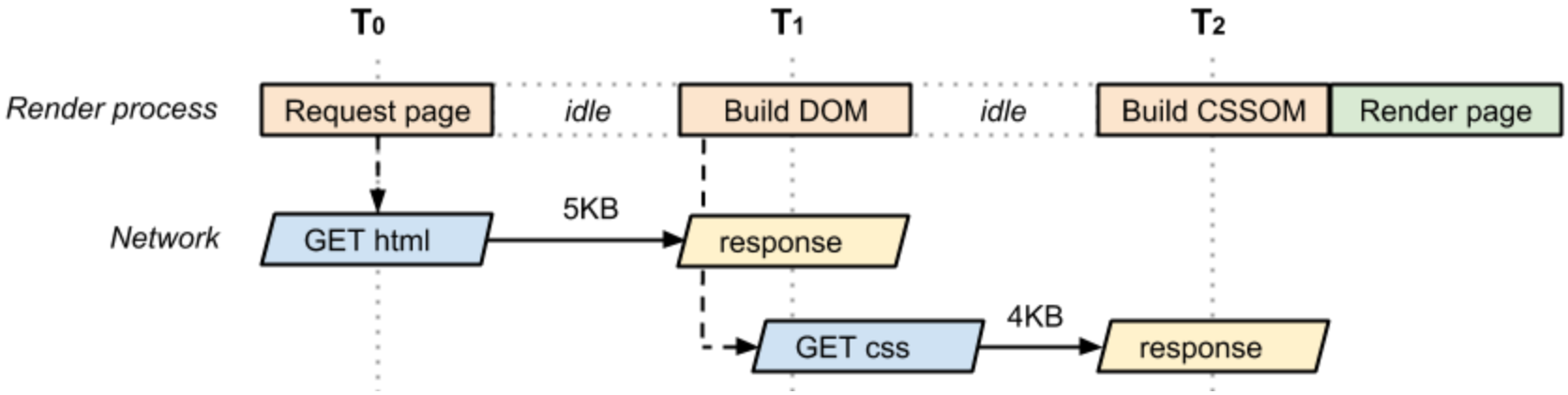


One critical resource
One round trip
5KB critical bytes

Page with external CSS

```
<html>
<head>
<title>Critical Path: Measure Script</title>
 <meta name="viewport" content="width=device-width,initial-scale=1">
<link href="style.css" rel="stylesheet">
</head>
<body onload="measureCRP()">
<p>Hello <span>web performance</span> students!</p>
 <div><img src="awesome-photo.jpg"></div>
</body>
</html>
```

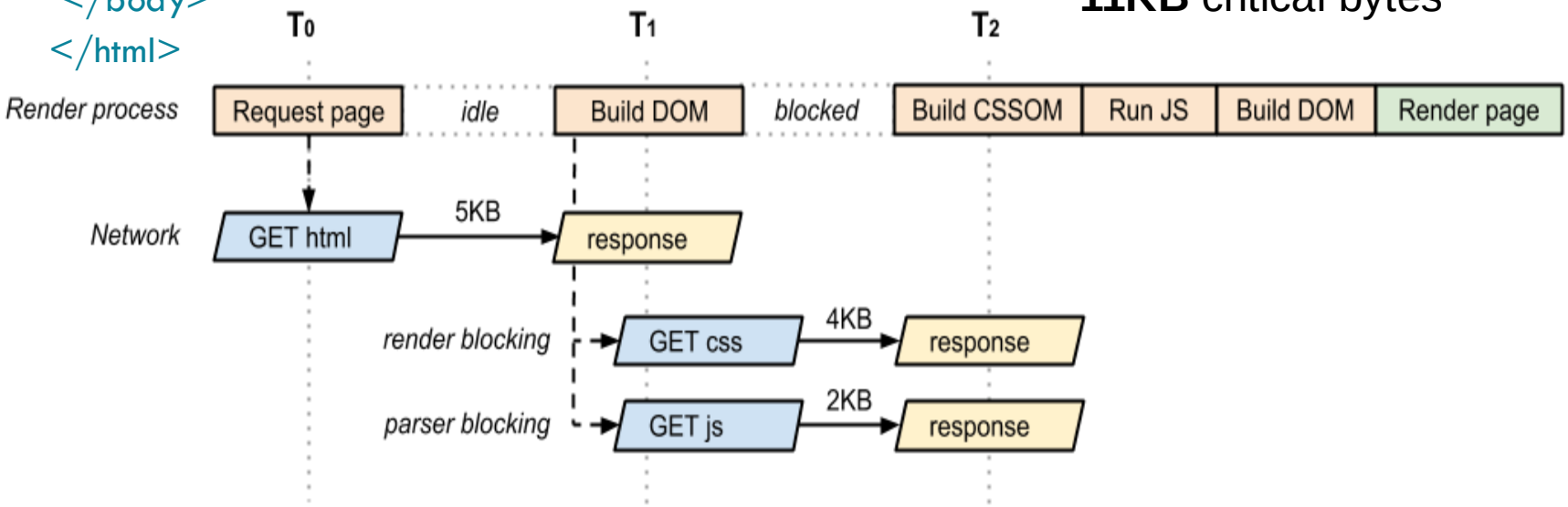
Two critical resources
Two round trips
9KB critical bytes



Page with external CSS and JS

```
<html>
<head>
<title>Critical Path: Measure Script</title>
 <meta name="viewport" content="width=device-width,initial-scale=1">
<link href="style.css" rel="stylesheet">
</head>
<body>
<p>Hello <span>web performance</span> students!</p>
 <div><img src="awesome-photo.jpg"></div>
<script src="app.js"></script>
 </body>
</html>
```

Three critical resources
Two round trips
11KB critical bytes



CRP Optimization

- Variables influence the CRP
  - Number of critical resources
  - Critical path length
  - Number of critical bytes

- General guidelines to optimize the CRP
  - Analyse and characterize the critical path
  - Minimize number of critical resources (defer, eliminate, async)
  - Optimize the number of critical bytes to reduce download time
  - Optimize the order in which the remaining critical resources are loaded

References

- Randy Connolly, Ricardo Hoar, Fundamentals of Web Development, Global Edition, Pearson

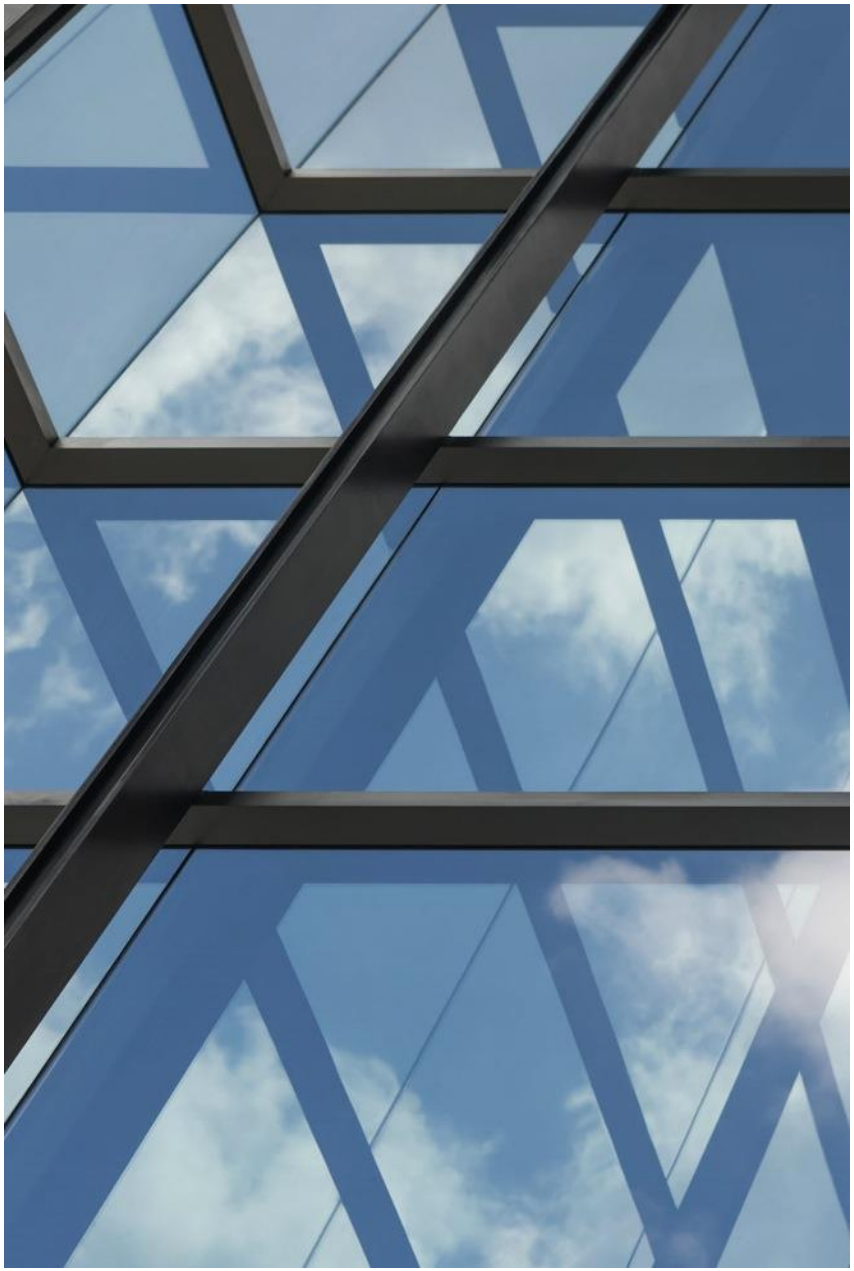
- W3Schools, JavaScript tutorial [https://www.w3schools.com/js/default.asp]

- Ilya Grigorik, Google Developers Web Fundamentals[https://developers.google.com/web/fundamentals/?hl=en]
  - Critical Rendering path

  [https://developers.google.com/web/fundamentals/performance/critical-rendering-path/?hl=en]

W4 Tutorial: The Browser tutorial

Week 5 Lecture: Server-side Development



THE UNIVERSITY OF SYDNEY