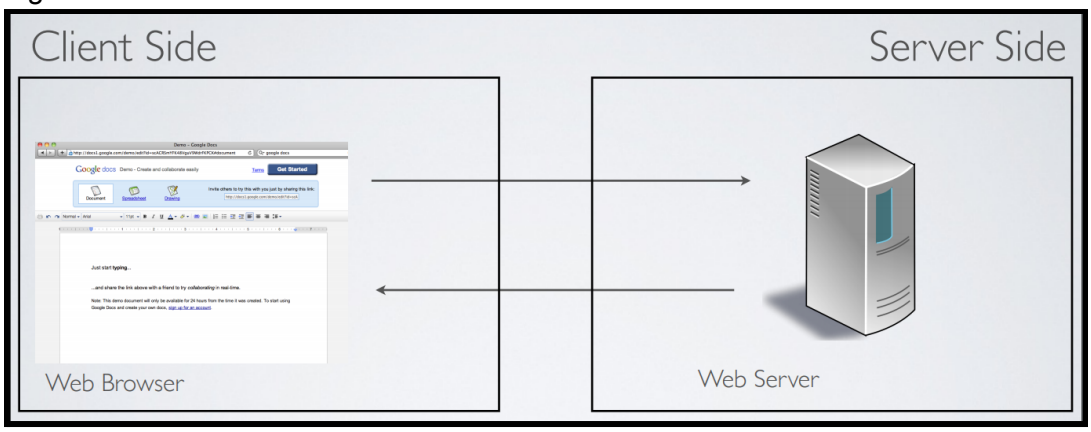
Introduction:

- A **web application** is a computer program that utilizes web browsers and web technology to perform tasks over the Internet.
- There's 2 parts to a web application:
    1. **Client side:**
    - The front end.
    - Runs on your browser on your machine (laptop, PC, phone, tablet, etc).
    2. **Server side:**
    - The back end.
    - Runs on a server.

    E.g.



The frontend and backend communicates with each other and sends data back and forth.

- Examples of frontend languages/markup language:
    - HTML (Markup language)
    - CSS (Markup language)
    - Javascript (Programming language)
- Examples of backend languages:
    - Python
    - Java
    - Go
    - NodeJS
- Web development has changed and evolved tremendously in the past few decades.
- One consequence of this evolution is that the application is moving from the server to the client. This makes for richer content. The traditional web platform was browsers, but now, modern web platforms are smartphones and tablets.
- A second consequence is cloud computing. Data storage and data processing are moving from the desktop to the cloud.
- HTML deals with content, while CSS deals with presentation and style and Javascript deals with processing.

HTML:

- Stands for Hyper-Text Markup Language.
- HTML is the standard markup language for creating Web pages.
- HTML describes the content and structure of a Web page.
- HTML consists of a series of elements.
- HTML elements tell the browser how to display the content.
- HTML is a markup language. It is not a programming language.

- **History of HTML:**

HTML 1	Invented by Tom Berners-Lee (1991)
HTML 2	First standard from W3C (1995)
HTML 4	Separation between content and presentation CSS (1997)
HTML 5	Multimedia (2008)

- **Terminology:**
- **Markup:** Tags starting and ending with angle brackets.
  E.g. ** **
  Note: Some tags have a start and end tag, while others only have the starting tag.
  E.g.
  ** ** ← Has both start and end tags. All end tags have a / in them.
  <img> ← Only has start tag.
- **Content:** Everything else.
- **Element:** A start and end tag and the content in between.
  I.e. An HTML element is defined by a start tag, some content in between the tags, and an end tag.
  E.g. ** Some content ** ← This entire thing is an element.
  Note: Some HTML elements have no content. They are called **empty elements**.
  E.g. The **
** tag defines a line break, and is an empty element without a closing tag.
- **Attribute:** The name/value pairs specified in a start tag.
  All HTML elements can have attributes.
  Attributes provide additional information about elements.
  Attributes are always specified in the start tag.
  E.g. Consider the **<a>** tag. The href attribute specifies the URL of the page the link goes to.
  I.e. ** … ** ← Notice how the href attribute is in the starting tag.
- **Comments:** Tags that will be ignored at rendering. In HTML, comments are denoted using **<!-- -->**.
  E.g. **<!-- This is a comment. -->**
- **Skeleton of a HTML document:**

```
<!DOCTYPE HTML>
<html>
        <head>
                This is where page metadata and invisible content goes.
                Anything you want to show on the website should not go here.
        </head>

        <body>
                This is where visible page content goes.
                I.e. Anything you want to show on the website should go here.
        </body>
</html>
```

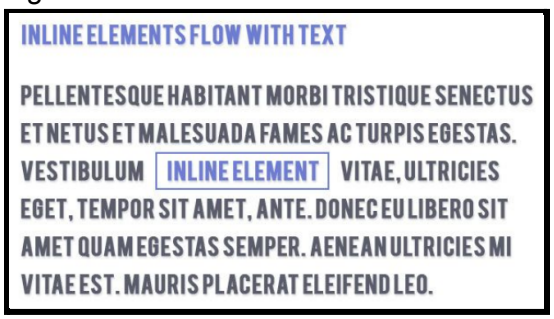
- **Differences between XHTML and HTML:**
- At first, browsers were quite forgiving about missing/mismatched tags in HTML, however, different browsers rendered differently.
- Then, HTML became XHTML (aka HTML 4 and 5).

- Now, there is homogeneity across browsers for the most part and it is easier to parse in Javascript. Furthermore, the styling disappeared in HTML 4 to become CSS and elements related to styling became deprecated. Some of these elements include **<b>**, **<i>**, **<font>**, etc.
  Note: The browser will still render them and not tell you that they're deprecated, but you should not use them.
- **Note:** If there are errors or missing elements/tags, the browser will not give an error. The browser will still try to load the HTML page.
- You can check if HTML is valid using **https://validator.w3.org/**.
- **List of HTML Tags:**
- **<!DOCTYPE html>:** All HTML documents must start with a <!DOCTYPE html> declaration. It is used to tell the browser about what document type to expect.
- **<a>:** The <a> tag defines a hyperlink, which is used to link from one page to another. The most important attribute of the <a> element is the href attribute, which indicates the link's destination.
  By default, links will appear as follows in all browsers:
  An unvisited link is underlined and blue.
  A visited link is underlined and purple.
  An active link is underlined and red.
  Note: If the <a> tag has no href attribute, it is only a placeholder for a hyperlink.
- **<body>:** The <body> tag defines the document's body. It contains all the contents of an HTML document, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.
  Note: There can only be one <body> element in an HTML document.
- ** :** The tag defines a division, block or a section in an HTML document.
  The tag is used as a container for HTML elements, which is then styled with CSS or manipulated with JavaScript.
  The tag is easily styled by using the class or id attribute.
  Any sort of content can be put inside the tag.
  Note: By default, browsers always place a line break before and after the element.
- **<h1> - <h6>:** The <h1> to <h6> tags are used to define HTML headings.
  <h1> defines the most important heading. <h6> defines the least important heading.
  Text in <h1> tags are larger than text in <h2> tags, and so on.
  Text in <h6> tags are the smallest.
- **<head>:** The <head> element is a container for metadata and is placed between the <html> tag and the <body> tag. Metadata is data about the HTML document. It is not displayed. Metadata typically defines the document title, character set, styles, scripts, and other meta information. You would put things like links to external CSS or Javascript files, the title tag among other stuff in the head tag.
- **<html>:** The <html> tag represents the root of an HTML document. The <html> tag is the container for all other HTML elements except for the <!DOCTYPE> tag.
  Note: You should always include the lang attribute inside the <html> tag, to declare the language of the webpage. This is meant to assist search engines and browsers.
- **<link>:** The <link> tag defines the relationship between the current document and an external resource. It is most often used to either contain internal CSS or link to external CSS file(s).
  Note: The <link> element is an empty element. It contains attributes only.
- **<p>:** The <p> tag defines a paragraph. Browsers automatically add a single blank line before and after each <p> element.
- **<script>:** The <script> tag is used to embed JavaScript. It either contains internal Javascript, or it points to an external script file through the src attribute.

- **:** The tag is an inline container used to mark up a part of a text, or a part of a document. It is easily styled by CSS or manipulated with JavaScript using the class or id attribute. The tag is much like the element, but is a block-level element and is an inline element.
- **<title>:** The <title> tag defines the title of the document. The title must be text-only, and it is shown in the browser's title bar or in the page's tab. It is required in HTML documents. The contents of a page title is very important for search engine optimization. The page title is used by search engine algorithms to decide the order when listing pages in search results.
  The <title> element:
  - Defines a title in the browser toolbar.
  - Provides a title for the page when it is added to favorites.
  - Displays a title for the page in search-engine results.

  Note: You can not have more than one <title> element in an HTML document.
- **Semantic vs Non-Semantic Tags:**
- **Semantic tags** are HTML tags that indicate their expected use to both the user and the browser. Examples include:
  <form>
  <h1>, <h2>, <h3>, <h4>, <h5>, <h6>
  <table>
- Some advantages of semantic tags are:
  1. **Accessibility:** Screen readers can change voice tone on a tag.
     E.g. If a word is in a tag, the screen reader would read that word louder.
  2. **Search Engine Optimization:** Density of keywords is higher when more semantic tags are used.
- **Non-semantic tags** tell nothing about its content. They are generic and have no specific purpose. Examples of non-semantic tags are and . is a generic inline element while is a generic block element. They are used more for creating natural divisions throughout your page.
- **Class vs Id:**
- **Class:**
  - The HTML class attribute is used to specify a class for an HTML element.
  - Multiple HTML elements can share the same class name.
  - The class attribute is often used to point to a class name in a style sheet. It can also be used by JavaScript to access and manipulate elements with the specific class name.
- **Id:**
  - The HTML id attribute is used to specify a unique id for an HTML element.
  - You cannot have more than one element with the same id in an HTML document. **Note:** If you do have more than one element with the same id in an HTML document, the browser won't raise an error and won't complain. However, it is incorrect to have more than 1 element with the same id.
  - The id attribute is used to point to a specific style declaration in a style sheet. It is also used by JavaScript to access and manipulate the element with the specific id.
- **Block vs Inline vs Inline-Block:**
- Elements on a web page can be displayed in different ways.
  1. **Inline:**
     - Examples include: , <a>,
,
     - It doesn't have defined width/height and can't have block elements inside it. It also doesn't create newlines.

- This is often used for changing part of a text inside a paragraph.
- An inline element does not start on a new line and it only takes up as much width as necessary.
- E.g.

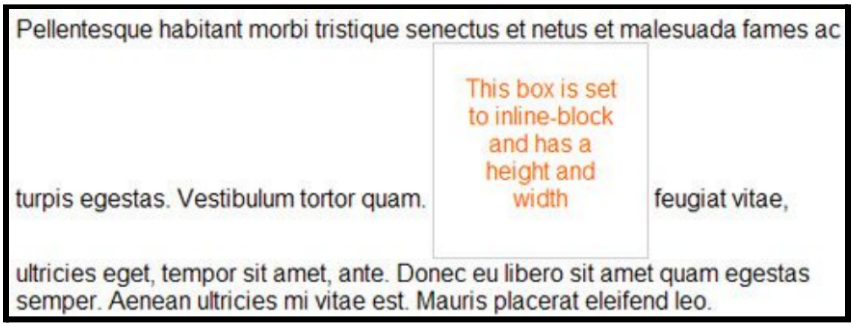


2. Block:

- Examples include: <p>, <h1> - <h6>,
- The height and width can be specified and changed. But by default, the width is the full width of the parent element and the height is enough to fit the content.
- It forces creation of newlines.
- A block-level element always starts on a new line and takes up the full width available. It stretches out to the left and right as far as it can.

3. Inline-block:

- Examples include: <img>
- These are inline elements that can have a height/width.



- **Div vs Span:**
- Both and are used to define parts of a web page.
- shows the inline portion of a document while shows a block-level portion of a document.
- The div should be used to wrap sections of a document, while spans are used to wrap small portions of text, images, etc.

CSS:

- Stands for Cascading Style Sheets.
- Previously, CSS was a part of HTML and we could style the webpage using HTML. However, that changed. One of the reasons it changed is because we could want the same style over multiple pages. Another reason is that across multiple platforms, we could want different layouts. For example, a website is not going to look the same for mobile and desktop. Lastly, we want to have custom layout for people with conditions. For example, some people are colour blind or they need to have large font and we want to let users be able to superimpose their own css to satisfy their needs.
- Comments in CSS are denoted with: **/* */**. Anything placed between /* and */ will be commented out.
- **CSS Format:**

```
selector{
        property: value;
        property: value;
        property: value;
}
```

- The selector points to the HTML element you want to style.
- The property: value pairs are usually fixed. This means that you have to use what is available and cannot create your own property: value pairs.
- E.g.
  Here, p is the selector, color and text-align are properties while red and center are values.

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

- There are a few ways we can write CSS selectors:
  1. **CSS element Selector:**
  - The element selector selects HTML elements based on the element name.
  - E.g.
    Here, all <p> elements on the page will be center-aligned, with a red text color.

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

  2. **CSS id Selector:**
  - The id selector uses the id attribute of an HTML element to select a specific element.
  - The id of an element is unique within a page, so the id selector is used to select one unique element.
  - To select an element with a specific id, write a hash character, #, followed by the id of the element.
  - E.g.
    The CSS rule below will be applied to the HTML element with id="para1".

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

3. **CSS class Selector:**

- The class selector selects HTML elements with a specific class attribute.
- To select elements with a specific class, write a period, ., followed by the class name.
- Unlike id's, you can use the same class name for multiple elements.
- E.g.
  In this example all HTML elements with class="center" will be red and center-aligned.

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

- **Note:** You can also specify that only specific HTML elements should be affected by a class.
- E.g.
  In this example only <p> elements with class="center" will be red and center-aligned.

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

4. **CSS Universal Selector:**

- The universal selector, *, selects all HTML elements on the page.
- E.g.
  The CSS rule below will affect every HTML element on the page.

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

5. **CSS Descendent Selector:**

- The descendant selector matches all elements that are descendants of a specified element. The first simple selector within this selector represents the parent element and the second simple selector represents the descendant element we're trying to match.
  E.g.

```
p strong{
        background-color: yellow;
}
```

  This applies to all elements that are inside a <p> element.

6. **CSS Multiple Element Selector:**

- The multiple element selector is used to select multiple, different elements.
  E.g.

```
p, strong, h1 {
        background-color: yellow;
}
```

  This applies to all <p>, and <h1> elements.

- **Conflicting Selectors:**
- When two selectors appear to conflict, the more specific selector takes precedence.
- E.g. Consider the code below:

HTML:

```
<p>
        <em> Green or red? </em>
</p>
```

CSS:

```
p em {
        color: green;
}

em {
        color: red;
}
```

Since “p em” is more specific than “em”, it will be used.

- **Specificity Precedence:**
    1. Ids (Highest specificity)
    2. Classes
    3. Elements (Lowest specificity)

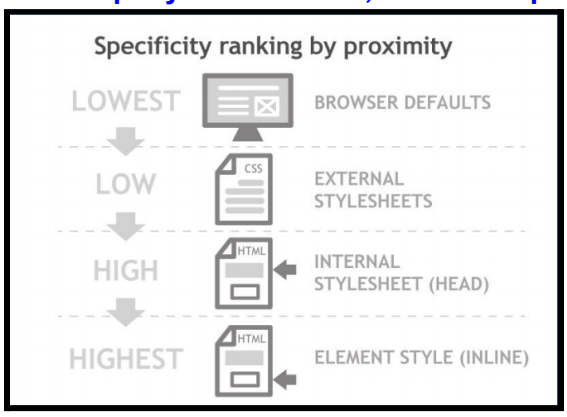
    Note: Parent elements are more specific than children elements.
- Furthermore, if you have multiple rules with the same specificity, then the rule further down the style sheet wins.
  E.g. If you have 2 rules about the <h1> tag, the rule that’s further down wins.
- **CSS Inheritance:**
- Children elements inherit parent styles in most cases.
- If the styles of a child HTML element are not specified, the element inherits the styles of its parent.
- **CSS - Inline, embedded or separate file:**
- There are three ways of inserting a style sheet:
    1. **External CSS:**
    - Here, you write the CSS in a separate file and save it as a .css file.
    - Each HTML page must include a reference to the external style sheet file inside the <link> element, inside the head section.
    - E.g.

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

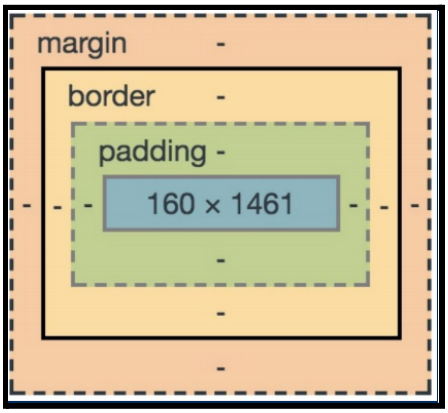
    2. **Internal CSS:**
    - You can include internal CSS inside the <style> element, inside the head section.
    - This way is not preferred.
    - E.g.

    ```
    <head>
            <style>
            … CSS stuff here ...
            </style>
    </head>
    ```

3. **Inline CSS:**
- An inline style may be used to apply a unique style for a single element.
- To use inline styles, add the style attribute to the relevant element. The style attribute can contain any CSS property.
- This way is also not preferred.
- E.g.
  ** This is a paragraph. **



- **CSS Box Model:**
- Think of each element as a box or rectangle.

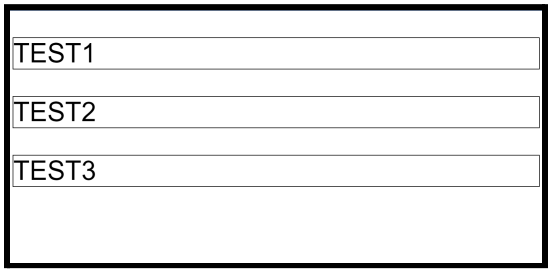


- The size of an element in the box model is determined by the blue rectangle (shown in the picture above) and the 3 rectangular rings around it (padding, border and margin).
- The blue rectangle in the center is the size of the content of the element.
  Note: The numbers are in pixels.
  We can modify the size of the content using the height and width properties in CSS.
- Just outside the content is the **padding**, which is the space between the content and the border of the element. It's used to give some space between the content and the border so that the content isn't touching the border.
- The **border** comes after the padding and it can be any size you want it to be.
- Lastly, we have the **margin**, which is the area around the border (clearing space). It's used so that the elements are not touching each other.
  I.e. The margin is used to give some space between elements.
- All size elements are properties in CSS.
- If the content is block-displayed, we can change its size using height and width. We can specify the content size using either the number of pixels or with a percentage of the parent element it is held in.

- E.g. Consider the code and picture below.

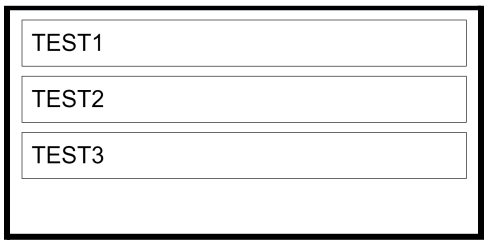
```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="test.css">
</head>

<body>
    <p> TEST1 </p>
    <p> TEST2 </p>
    <p> TEST3 </p>
</body>
</html>
```

```
p{
    border: 2px solid black;
    font-size: 5rem;
}
```



Now, if I add margin and padding, like shown below, notice how the area within each box and text increased, and the area between boxes and the area between a box and the edges all changed.

```
p{
    border: 2px solid black;
    font-size: 5rem;
    padding: 2%;
    margin: 2%;
}
```



- **CSS Positioning:**
- **Positioning of elements** refers to a deviation from their **natural flow**, which is where the elements are placed by default.
- Here are a few ways you can position an element using CSS:
    1. **Static:** The default position of the element.
       When you put an element in HTML without changing the position type, you are by default putting it into a static position.
    2. **Fixed:** It fixes the element in one position in the viewport of the browser.
       Elements in a fixed position do not move even with scrolling.
    3. **Relative:** Allows us to change the position of an element relative to its natural location.
       E.g. We can say that we want to move an element x pixels to the right of where it is naturally located.
    4. **Absolute:** An absolutely positioned element is positioned relative to the first positioned parent. However, if an absolute positioned element has no positioned parent, it uses the document body.
    5. **Sticky:** The element is treated like a relative value until the scroll location of the viewport reaches a specified threshold, at which point the element takes a fixed position where it is told to stick.
- **Note:** Except for position: absolute, if both top and bottom are specified and are not auto, top wins. If both left and right are specified and are not auto, left wins when direction is left to right (ltr) (English, horizontal Japanese, etc) and right wins when direction is right to left (rtl) (Persian, Arabic, Hebrew, etc). In the case of position: absolute, if you have both top and bottom, it will span the page. Same with left and right. Top, bottom, left, and right are css properties.
- **CSS Float:**
- The CSS float property places an element on the left or right side of its container, allowing text and inline elements to wrap around it.
- The float property is used for positioning and formatting content.
- The float property can have one of the following values:
    - **Left:** The element floats to the left of its container.
    - **Right:** The element floats to the right of its container.
    - **None:** The element does not float. It will be displayed just where it occurs in the text. This is default.
    - **Inherit:** The element inherits the float value of its parent.

Introduction to JavaScript:

- In 1995, the browser that dominated the market was Netscape Navigator. However, with only static HTML pages, they were eager to improve the experience. They decided they needed a scripting language that would let developers make the internet more dynamic. A Netscape employee named Brendan Eich made the first version in 10 days. After a few iterations, they named the language JavaScript since Java was popular back then. Otherwise the languages don't have much connection.
- Soon, **JavaScript (JS)** was becoming popular. A standard was created by ECMA for scripting languages. The standard was based on JS. **ECMAScript (ES)** is a scripting-language specification standardized by Ecma International. It was created to standardize JavaScript to help foster multiple independent implementations. ES is the standard, while JS implements that standard.
  Note: JavaScript's official name is ECMAScript.
- As time went on, ES's standards improved.
- ES3 (1999) is the baseline for modern day Javascript.
- A bunch of others, like Mozilla, started to work hard on ES5, which was released in 2009.
- Although more versions of the standard have been released, we will mostly talk about new features up to ES6 (2015).
- Browsers adopt new ES standards slowly, but ES6 is mostly completely adopted by modern browsers.
- JavaScript is the programming language of HTML and the Web.
- While HTML is used to define the content of web pages and CSS is used to specify the layout of web pages, JavaScript is used to program the behavior of web pages.
- Javascript is an interpreted programming language that is used to make webpages interactive.
- It's object based.
- It runs on the client's browser.
- It uses events and actions to make webpages interactive.
- JavaScript is a loosely typed and dynamic language. Variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned and re-assigned values of all types.
- JavaScript is also functional. This means it has **first-class functions**. A programming language is said to have first-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.
- **Note:** Javascript has absolutely nothing to do with Java. It was called JavaScript because at the time it was created, Java was extremely popular.
  "Java is to Javascript is what a car is to a carpet"

Basic JavaScript:

- **Comments:**
- Single line comments are denoted by **//**.
- Multi-line comments are denoted by **/* */**.
- **Linking JavaScript in HTML:**
- There are 3 ways to link JavaScript in HTML:
    1. **Inline:**
    - Here, JavaScript is written in the HTML page but without using the <script> tag.
    - E.g. **<button onclick="console.log("Hello World!);">Click me</button>**
    2. **Embedded:**
    - Here, JavaScript is written in the HTML page in a <script> tag.
    - E.g.

```
<script type="text/javascript">
        console.log("Hello World!);
</script>
```

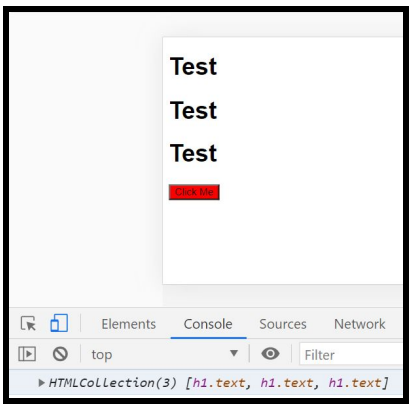
    3. **External:**
    - Here, JavaScript is written in an external .js file and linked to a HTML page using the <script> tag.
    - E.g. **<script src="js/script.js"></script>**
- **Output:**
- JavaScript can output data in different ways:
    1. Writing into an HTML element, using **innerHTML**.
    2. Writing into the HTML output using **document.write()**.
    3. Writing into a pop-up box, using **window.alert()** or **confirm()** or **prompt()**.
    4. Writing into the browser console, using **console.log()**.
- Using innerHTML:
- To access an HTML element, JavaScript can use the **document.getElementById()** method. This is the most common way of obtaining an element to modify. Since ids are unique, we can access a pre-existing element in HTML using this strategy.
- The id attribute defines the HTML element. The innerHTML property defines the HTML content.
- E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            button.style.backgroundColor = "red";
            document.getElementById("text").innerHTML = "Test";
        }
    </script>
</head>

<body>

    <h1 id="text"> Sample Text </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```

Originally, the page displays “Sample Text”, but after you click on the button, it changes to “Test” because I modified the value of id=“demo” using **document.getElementById("text").innerHTML**.

- **Note:** There is a similar command called **document.getElementsByClassName()**. However since classes are not unique, this command will always return the result in an array. Hence, to change all elements with the given class, you need to loop through the array.
- E.g. Consider the code and output below:

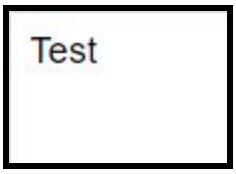
```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            button.style.backgroundColor = "red";
            let x=document.getElementsByClassName("text");  // Find the elements
            console.log(x);
            for(var i = 0; i < x.length; i++){
                x[i].innerText="Test";    // Change the content
            }
        }
    </script>
</head>

<body>

    <h1 class="text"> Sample Text 1 </h1>
    <h1 class="text"> Sample Text 2 </h1>
    <h1 class="text"> Sample Text 3 </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```



Notice that **x=document.getElementsByClassName("text");** gets you an array. I.e. x is an array.
You can see that x is an array from the bottom right picture on the console. Furthermore, I had to iterate through x to change the elements.

- Using document.write():
- document.write() lets you write onto HTML output.
- **Note:** Using document.write() after an HTML document is loaded, will delete all existing HTML.
- E.g. Consider the code and output below:

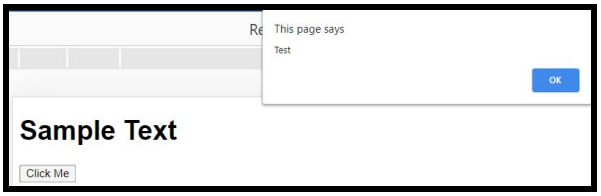
```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            document.write("Test");
        }
    </script>
</head>

<body>

    <h1 id="text"> Sample Text </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```



Notice how the HTML output was replaced with “Test” after clicking the button.

- Using window.alert(), prompt() and confirm():
- You can use alert() to create an alert box to display data.
- E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            // button.style.backgroundColor = "red";
            alert("Test");
        }
    </script>
</head>

<body>

    <h1 id="text"> Sample Text </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```



Notice that after clicking on the button, the pop-up (alert box) appears.

- E.g. For confirm()

```
<script type="text/javascript">
    confirm("Are you sure?");
</script>
```

Confirm() creates a dialog box with “ok” and “cancel” buttons.

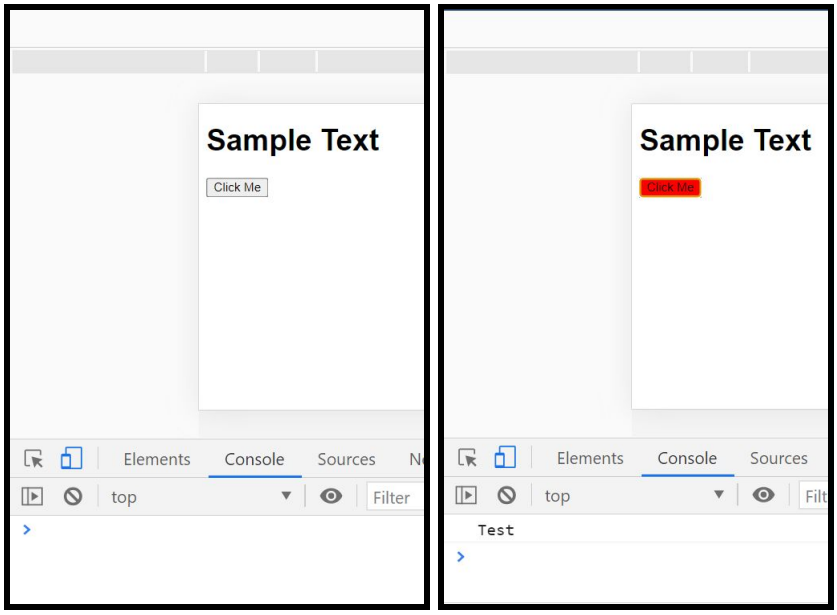
- E.g. For prompt()

```
<script type="text/javascript">
    prompt("Name?","John");
</script>
```

The prompt() method displays a dialog box that prompts the visitor for input.
A prompt box is often used if you want the user to input a value before entering a page.
You can give a default value. In the example above, the default value is John.

- Using console.log():
- For debugging purposes, you can use the console.log() method to print stuff in the console.
- E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            button.style.backgroundColor = "red";
            console.log("Test");
        }
    </script>
</head>

<body>

    <h1 id="text"> Sample Text </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```



Originally, there's nothing in the console log, but when you click on the button you'll see "Test" there.

- **Scoping:**
- **Scope** determines the visibility or accessibility of a variable or other resource in the area of your code.
- If a variable is declared outside a function, it has **global scope**. This means that all functions can access it.
- Variables declared inside the functions become local to the function and are considered in the corresponding **local scope**. Every function has its own scope. The same variable can be used in different functions because they are bound to their respective functions and are not mutually visible. Local scope can be divided into **function scope** and **block scope**. The concept of block scope was introduced in ES6 together with const and let.
- Whenever you declare a variable in a function, the variable is visible only within the function. This is **function scope**. You can't access it outside the function. var has function scope.
- If a variable is declared inside an if statement, switch statement, for or while loop, it has **block scope**. It can only be accessed inside that block.
- **Lexical scope** means the children scope has access to the variables defined in the parent scope.
  E.g. If you have a nested function, the inner function can still access the variables declared in the outer function.
  E.g. If you have an if statement, switch statement, for or while loop in a function, the statement or loop can still access the variables outside it but still within the function.
- **Constants:**
- To declare constants in JavaScript, we use the **const** keyword.
- The value of a constant can't be changed through reassignment, and it can't be redeclared.
- **Note:** The const declaration creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned. For instance, in the case where the content is an object, this means the object's contents, its properties, can be altered.

- An initializer for a constant is required. You must specify its value in the same statement in which it's declared. This makes sense, given that it can't be changed later.
- Const has block scope.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    const x = "Hello";
    console.log(x);
    x = "Hi"; // Will give error
</script>
```

```
Hello
▶Uncaught TypeError: Assignment to constant variable.
    at test.html:8
```

Notice that when I try to reassign x, I get an error.

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    const x = [1,2,3,4,5,6];
    console.log(x);
    x[0] = 0; // Will not give error
    console.log(x);
</script>
```

```
▶ (6) [1, 2, 3, 4, 5, 6]
▶ (6) [0, 2, 3, 4, 5, 6]
```

In this case, because x is an array, we can change its contents.
However, if you try to change x to an empty array, you'll get an error.

```
<script type="text/javascript">
    const x = [1,2,3,4,5,6];
    console.log(x);
    x = []; // Will give an error.
</script>
```

```
▶ (6) [1, 2, 3, 4, 5, 6]
▶Uncaught TypeError: Assignment to constant variable.
    at test.html:8
```

- **Variables:**
- In JS, we can declare variables using the **var** or the **let** keyword.
- A variable declared without a value will have the value undefined.

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x;
    console.log(x);
</script>
```

```
undefined
>
```

```
<script type="text/javascript">
    var x;
    console.log(x);
</script>
```

```
undefined
>
```

- Variables declared using var have **function scope** while variables declared with let have **block scope**. For this reason, it is better to use let than var.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    if (1 < 3){
        var x = 5;
    }
    console.log(x);
</script>
```

```
5
>
```

Notice that even though x was created in the if statement, we could still access it outside.

```
<script type="text/javascript">
    if (1 < 3){
        let x = 5;
    }
    console.log(x);
</script>
```

```
Uncaught ReferenceError: x is not defined
    at test.html:9
```

Notice that when we use let, we can no longer access x outside of the if statement. This is because let has block scope while var has function scope.

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    console.log(x);
    if (1 < 3){
        var x = 5;
    }
</script>
```

```
undefined
>
```

Notice that even though I didn't declare the variable x, console.log(x) still prints undefined. However, if we change the var to let, shown below, it throws an error. This is due to hoisting.

```
<script type="text/javascript">
    console.log(x);
    if (1 < 3){
        let x = 5;
    }
</script>
```

```
Uncaught ReferenceError: x is not defined
    at test.html:6
```

- **Note:** Even though it's better practice to use let, some old browsers can only handle var.
- **Hoisting:**
- **Hoisting** is JavaScript's default behavior of storing function and variable declarations into memory first.
- **Note:** JavaScript only hoists declarations, not initializations.
- When you run your JavaScript code, the function and variable declarations are put into memory first. Specifically, the variable and function declarations are put into memory during the compile phase.
- **Note:** Functions are hoisted before variables.
- Conceptually, you can think about hoisting as JavaScript's behavior of moving declarations to the top of its scope.
- Since variables declared with var have function scope, if we call a variable declared using var before initializing it, we get undefined.
- E.g. Consider the code and output below:

```
function f1(){
    console.log(a)
    var a = 3
}

f1()
```

```
undefined
```

What's happening is this:

```
function f1(){
    var a // Declaring "a" at the top of the function.
    console.log(a) //While a has been declared, it hasn't been defined, so its value is undefined
    a = 3 // Definition stays in place.
}
f1()
```

- However, if you try to call a variable/constant declared with let or const, respectively, we get an error. This is because let and const have block scope and are initialized at runtime. Furthermore, let and const variables/constants cannot be read/written until they have been fully initialized. Accessing the variable before the initialization results in a ReferenceError. The variable is said to be in a temporal dead zone from the start of the block until the initialization has completed.

- **Use Strict:**
- The keyword **"use strict"** defines that JavaScript code should be executed in "strict mode".
- It helps you to write cleaner code, such as preventing you from using undeclared variables.
- Strict mode is declared by adding **"use strict"** to the beginning of a script or a function.
- If it is declared at the beginning of a script, it has global scope. If it is declared inside a function, it has local scope.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    "use strict"
    y = 4;
    f1();
    function f1(){
        x = 3.14;
    }
</script>
```

```
Uncaught ReferenceError: y is not defined
    at test.html:7
```

Here, since we used “use strict” at the top of the script, it has global scope.

```
<script type="text/javascript">
    y = 4;
    f1();
    function f1(){
        "use strict"
        x = 3.14;
    }
</script>
```

```
Uncaught ReferenceError: x is not defined
    at f1 (test.html:10)
    at test.html:7
```

Here, since we used “use strict” in f1, it has local scope and as a result, y = 4 isn’t affected.

- Strict mode makes it easier to write better, cleaner and more secure JavaScript.
- Strict mode changes previously accepted bad syntax into real errors.

- **Arrays:**
- Can be declared in multiple ways:
    1. **let myArray = new Array();**
       myArray[0] = "JavaScript";
       myArray[1] = "is";
       myArray[2] = "fun";
    2. **let myArray = new Array ("Javascript","is","fun");**
    3. **let myArray = ["Javascript","is","fun"];**
- Array indexes are 0 based, meaning that array[0] is the first element of the array.
- **Data and Structure Types:**
- Because JavaScript is a loosely typed and dynamic language, it has dynamic data types. This means that the same variable can be used to hold different data types. Furthermore, JavaScript does not need you to specify the type of variables.
- You can use the **typeof** operator to get the type of any variable.
- A **primitive data value** is a single simple data value with no additional properties and methods.
- There are six data types that are primitives:
    1. **Undefined:**
    - In JavaScript, a variable without a value has the value **undefined**. The type is also undefined.
    - E.g. **let car; // Value is undefined, type is undefined**
    - Any variable can be emptied, by setting the value to undefined. The type will also be undefined.
    - E.g.
      let person = 123;
      person = undefined; // Now both value and type is undefined
    - typeof instance == “undefined”
    2. **Boolean:**
    - Booleans can only have two values: true or false.
    - typeof instance == “boolean”
    3. **String:**
    - typeof instance == “string”
    4. **Number:**
    - typeof instance == “number”
    5. **BigInt:**
    - In JavaScript, BigInt is a numeric data type that can represent integers in the arbitrary precision format.
    - typeof instance == “number”
    6. **Symbol:**
    - A value having the data type Symbol can be referred to as a "Symbol value". In a JavaScript runtime environment, a symbol value is created by invoking the function Symbol, which dynamically produces an anonymous, unique value. A symbol may be used as an object property.
    - typeof instance == “symbol”

- There are 2 structural types:
    1. **Object:**
    - JavaScript objects are written with curly braces {}.
    - JavaScript objects can be declared in multiple ways:
        1. **let myDict = new Object();**
           myDict["first"] = "JavaScript";
           myDict["second"] = "is";
           myDict["third"] = "fun";
        2. **let myDict = {};**
           myDict.first = "JavaScript";
           myDict.second = "is";
           myDict.third = "fun";
        3. **let myDict = {first: "Javascript", second: "is", third: "fun"}**
    - **Note:** The typeof operator returns object for objects, arrays, null, etc.
    - An object in JavaScript is simply a set of key-value pairs.
    - Keys are called **properties**. They must be strings.
    - You can access object properties in two ways:

    1. objectName.propertyName

    2. objectName[“propertyName”]

    - Properties can be added and changed.
      Note: If you create an object using const, you can still change and/or add properties. You just can’t reassign the object.
    - Objects can also have **methods**. Methods are actions that can be performed on objects. A method is a function stored as a property. You access an object method with the following syntax: **objectName.methodName()**.
    - In JavaScript, almost everything is an object.
        - Booleans can be objects if defined with the new keyword.
        - Numbers can be objects if defined with the new keyword.
        - Strings can be objects if defined with the new keyword.
        - Arrays are always objects.
        - Objects are always objects.

    2. **Functions:**
    - typeof instance == “function”
- There is 1 structural root primitive type:
    1. **Null:**
    - In JavaScript null is nothing. It is supposed to be something that doesn't exist.
    - In JavaScript, the data type of null is an object.
    - You can empty an object by setting it to null.
    - E.g.
      let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
      person = null; // Now value is null, but type is still an object
    - Undefined and null are equal in value but different in type.
    - typeof instance == “object”

- **If Statements:**
- Syntax:

```
if (condition){
 // Code inside the if block.
}
else if (condition){
 // Code inside the else if block.
}
else {
 // Code inside the else block.
}
```

- **For loops:**
- Syntax:

```
for (statement 1; statement 2; statement 3){
 // Code to run inside loop
}
```

- E.g.

```
for (let i = 0; i < 5; i++){
   console.log(i);
}
```

- **For In Loops:**
- The JavaScript for-in statement loops through the properties of an object that are keyed by strings.
- Syntax:

```
for (variable in object){
 // Code to run inside loop
}
```

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = {'a':1, 'b':2}; // A JavaScript object.
    for (const element in x){
        console.log(element);
    }
</script>
```

```
a
b
>
```

- **For/Of Loop:**
- The JavaScript for/of statement loops through the values of an iterable objects such as Arrays, Strings, Maps, NodeLists, and more.
- **Note:** A JavaScript object is not iterable.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = {'a':1, 'b':2}; // A JavaScript object.
    for (const element of x){
        console.log(element);
    }
</script>
```

```
▶Uncaught TypeError: x is not iterable
    at test.html:7
```

- Syntax:

```
for (variable of iterable) {
  // Code to run inside loop
}
```

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = [1, 2, 3, 4, 5];
    for (const element of x){
        console.log(element);
    }
</script>
```

```
1
2
3
4
5
>
```

- **While loops:**
- Syntax:

```
while (condition) {
  // Code to run inside loop
}
```

- E.g.

```
let i = 1;
while (i < 10){
    console.log(i);
    i++;
}
```

- **Do/While Loop:**
- The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.
- Syntax:

```
do {
  // Code to run inside loop
}
while (condition);
```

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let i = 1;
    do{
        console.log(i);
        i++;
    }
    while (i < 0)
</script>
```

```
1
>
```

- **Note:** Because the condition is at the end, the loop will always run at least once.
- **Functions:**
- A **function** is a block of code designed to perform a particular task.
- A function is executed when something invokes it.
- Syntax:

```
function function_name(parameter1, parameter2, …, parameterN){
 // Code inside the function
}
```

- To invoke a function, put the name of the function, followed by parentheses () and with all the parameters inside the ().
  Note: Accessing a function without the () will return the function object instead of the return value.
- E.g. Consider the code and output below:

```
"use strict"
function f1(a, b, c){
    return(a+b+c)
}
console.log(f1(1, 2, 3)) // Prints the return value of f1.
console.log(f1) // Prints the function object.
```

```
6
ƒ f1(a, b, c){
    return(a+b+c)
}
```

- Because functions in JavaScript are first-class functions, they can be passed around without names. These are known as **anonymous functions** or **lambda functions**.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = function (a, b,){
        return a + b;
    }
    console.log(x(1, 2));
</script>
```

```
3
>
```

- Anonymous functions will be very useful for object methods and callback methods.
- With ES6, **arrow functions** were introduced to allow us to write shorter function syntax.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = (a, b) => {return a+b;}
    console.log(x(1, 2));
</script>
```

```
3
>
```

  This does the same thing as the function in the previous example above, but is much shorter to write.
- Beyond anonymous functions and arrow functions, a function can be passed as an argument to other functions, and can be returned by another function.
- **Equality:**
- There are two main ways of checking for equality in Javascript. We can use either the == operator or the === operator. However, the == operator will sometimes produce odd results. The === operator checks for type equality as well as content equality. A strict comparison (===) is only true if the operands are of the same type and the contents match. However, (==) converts the operands to the same type before making the comparison. Therefore, it is recommended that you only use the === operator.

- E.g.
  console.log(1 == 1); // true
  console.log('1' == 1); // true
  console.log(1 === 1); // true
  console.log('1' === 1); // false
- **Table of JavaScript Arithmetic Operators:**

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

- **Table of JavaScript Assignment Operators:**

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

- **Table of JavaScript Comparison Operators:**

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

- **Table of JavaScript Logical Operators:**

Operator	Description
&&	logical and
||	logical or
!	logical not

- **Table of JavaScript Type Operators:**

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Object Oriented Programing in JavaScript:

- **This:**
- The JavaScript keyword **this** refers to the object it belongs to.
- It has different values depending on where it is used:

Item	**What this refers to**
In the global execution context I.e. Outside of any function.	Refers to the global variable. In most browsers, the global variable is Window. **Note: this** will refer to the global object whether in strict mode or not.
In a method I.e. A method is a function in an object.	Refers to the owner object.
In a function without “Use Strict”	Refers to the global variable. In most browsers, the global variable is Window.
In a function with “Use Strict”	Refers to undefined.
In a constructor function I.e. When a function is invoked with the **new** keyword, then the function is known as a **constructor function** and returns a new instance.	Refers to the newly created instance.
In an event	Refers to the element that received the event.

- E.g.
  Here, we have a standalone **console.log(this);** outside of any function. The value of **this** refers to the global object, which is Window.

```
console.log(this);
```

```
▶ Window {window: Window, self: Window, document: document, name: "", location: Location, …}
```

Now, if we use “use strict”, we get the same result.

```
"use strict"
console.log(this);
```

```
▶ Window {window: Window, self: Window, document: document, name: "", location: Location, …}
```

If we use **this** in a method, **this** refers to the owner object.

```
let x = {
    name: "Rick",
    f1: function(){
        return this;
    }
}

console.log(x.f1())
```

```
▶ {name: "Rick", f1: f}
```

Here, the owner object is x, so, **this** represents the object x.

If we use **this** in a function without “use strict”, it will refer to the global variable.

```
(function f1(){
    console.log(this);
})();
```

```
▶ Window {window: Window, self: Window, document: document, name: "Test", location: Location, …}
```

If we use **this** in a function with “use strict”, it will refer to undefined.

```
(function f1(){
    "use strict";
    console.log(this);
})();
```

```
undefined
>
```

- **New:**
- The **new** operator lets developers create an instance of a user-defined object type or of one of the built-in object types that has a **constructor function**.
- 4 things that **new** does (in order):
    1. Creates a new empty object.
    2. Sets the new object’s delegate prototype (the object’s **__proto__**) to the constructor’s prototype property value.
    3. Calls the constructor function with this set to the new object.
    4. Returns the new object or **this**, if an object isn’t returned.
- **Constructor Function:**
- A **constructor function** is a function that creates a new object.
- It is often used with the **new** operator.
- E.g.

```
function person(name, age){
    this.name = name;
    this.age = age;
}

let Rick = new person("Rick", 21);
console.log(Rick);
```

```
▶ person {name: "Rick", age: 21}
```

Here, person is a constructor function. We’re creating a new person object with the **new** operator.

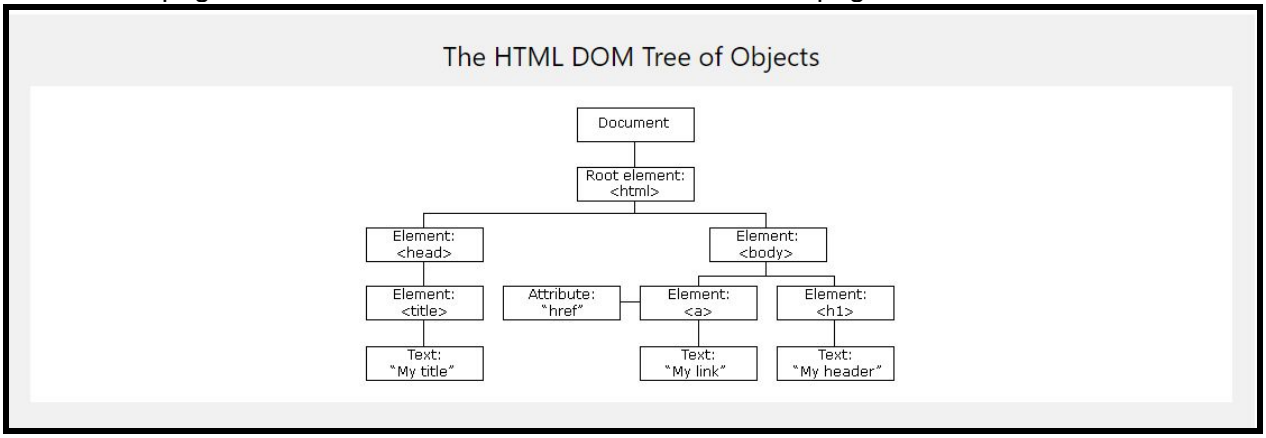
- **Note:** You cannot add a new property to an object constructor the same way you add a new property to an existing object. To add a new property to a constructor, you must add it to the constructor function.

JavaScript in the Browser:

- Javascript in the browser is restrictive.
- You can:
  - Access elements of the webpage and the browser.
  - Track user actions on the webpage (events).
  - Create threads (web workers).
  - Open sockets (web sockets).
- You cannot:
  - Access the file system. However, if you have an upload form, you can still access files uploaded.
  - Access to other programs.
  - Access to other tabs in the browser.
- **The Browser:**
- JavaScript can get you information about the user's browser, such as:

`screen`	the visitor's screen
`browser`	the browser itself
`window`	the current browser window
`url`	the current url
`history`	Back and forward URLs

- **DOM:**
- Stands for **Document Object Model**.
- When a webpage is loaded, the browser creates a DOM of the page.



- The entire document is a document node.
- Every HTML element is an element node.

- The text inside HTML elements are text nodes.
- Every HTML attribute is an attribute node.
- The root node is document.
- To access a node, you can use one of the following ways:
    - **document.getElementById("id");**
    - **document.getElementByTagName("p");**
    - **document.getElementByClassName("class");**
    - **document.querySelector("#id .class p");**
    - **document.querySelectorAll("#id .class p");**

  Note: When you use these DOM methods, you are manipulating/changing the DOM. For example, when you do x.style, you are manipulating the CSS of the DOM element x, but you are not making any changes to the css file itself.
- The nodes in the node tree have a hierarchical relationship to each other.
- The terms **parent**, **child**, and **sibling** are used to describe the relationships.
- In a node tree, the top node is called the **root** or **root node**.
- Every node has exactly one **parent**, except the root which has no parent.
- A node can have a number of children.
- **Siblings** are nodes with the same parent.
- You can use the following methods to navigate between nodes with JavaScript:
    - parentElement
    - children
    - firstElementChild
    - lastElementChild
    - nextElementSibling
    - previousElementSibling
- **Events:**
- An HTML **event** can be something the browser does, or something a user does.
- All events occur on HTML elements in the browser.
- JavaScript is used to define the action that needs to be taken when an event occurs. I.e. When [HTML Event] , do [JS Action]
- Here are some examples of HTML events:
    - An HTML web page has finished loading.
    - An HTML input field was changed.
    - An HTML button was clicked.
- An **event listener** in JS programmatically sets an event attribute on an HTML element.
  Syntax: **element.addEventListener(event, functionToExecuteWhenEventOccurs)**
  Note: functionToExecuteWhenEventOccurs is called a **callback function**.
- A **callback** is a function that is designated to be ‘called back’ at an appropriate time. In the case of events, it will be ‘called back’ when the event occurs. It can be an anonymous function, or a function defined outside of the event listener.
- There are 2 ways to call a function using addEventListener:
    1. **button.addEventListener(‘click’, function() { alert(‘Clicked’) });**
    2. **function alertClick() { alert(‘Clicked’) }**
       button.addEventListener(‘click’, alertClick);

  In the first way, you’re putting the function inside addEventListener and in the second way, you have an external function that you’re calling from the addEventListener.
- All events that occur create a JS Object with information about that event.
  Event.target gives information about the event origin element.
  Event.type gives information about the type of event.

- Events can be passed to the callback function as an argument.
  E.g.
  ```
  function myCallback(e) {
      // find information about e.
      // execute proper code.
  }
  ```
- We can use JavaScript to create events by using the dispatchEvent function.
- To use the dispatchEvent function, we have to create an Event first.
  We can do it in 2 ways:
  1. `let event = new Event(...);`
     `target.dispatchEvent(event);`
  2. `target.dispatchEvent(new Event(...));`
- E.g. Consider the code below:

```
<!DOCTYPE html>
<html>

<head>

</head>

<body>
    <button id="button" onclick="console.log('Clicked')"> Click Me </button>

    <script type="text/javascript">
        let event = new Event("click");
        button.dispatchEvent(event);

    </script>
</body>
</html>
```

Here, I am using the first way. Each time I refresh the page, I'll see Clicked in the console.

```
<!DOCTYPE html>
<html>

<head>

</head>

<body>
    <button id="button" onclick="console.log('Clicked')"> Click Me </button>

    <script type="text/javascript">
        button.dispatchEvent(new Event("click"));

    </script>
</body>
</html>
```

Here, I'm using the second way. It does the same thing as above.

- **Note:** addEventListener listens for an event while dispatchEvent creates an event.

Recipes to become a good front-end developer:

- Load Javascript code efficiently.
- Ensure the DOM is loaded with window.onload.
  Note: Suppose you either link an external JavaScript file in the head or you have embedded JavaScript before the <body> tag. Because the HTML file is loaded from top to bottom, if your JavaScript code needs to use something from the HTML code, like id or class, and you don't use window.onload, you'll get an error or an unexpected value. This is because the JavaScript code will get loaded before the HTML code, so the browser can't find the element(s) associated with the id or class.

  E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        let x = document.getElementById("button");
        console.log(x);
    </script>
</head>

<body>
    <button id="button"> Click Me </button>

</body>
</html>
```

```
null
>
```

Here, I'm trying to get the element with the id "button", which we have. However, because the JavaScript code is loaded before the HTML code, the browser can't find the element with the id "button" as it has not been created yet. Hence, console.log(x) gets us null.

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        window.onload = function(){
            let x = document.getElementById("button");
            console.log(x);
        }
    </script>
</head>

<body>
    <button id="button"> Click Me </button>

</body>
</html>
```

```
<button id="button"> Click Me </button>
>
```

Now, if we use window.onload, the browser will load the HTML code before loading the JavaScript code. As a result, we can get the element with the id “button” now.

- Encapsulate Javascript in closures.
- Create a Frontend API

The problem with Javascript interpreters:

- If you write good JavaScript code, it’ll be interpreted by different browsers in the same way. However, if you write bad or sloppy JavaScript code, it’ll be interpreted by different browsers in different ways. This is extremely bad.

How to write good JavaScript code:

1. Use “use strict”

- It forces the browser to validate Javascript against the standard and will dynamically raise errors or warnings in the console when the code is not compliant with the standard.
- See above on page 10.

2. Use JSHint

- Analyze Javascript source code with JSHint.
- JSHint will statically find bugs and report them in the terminal.

The problem with scoping:

- In the browser, all Javascript files share the same execution environment.
  I.e They share the same scope.
  E.g. Consider the code and output below:

```
<script src="test1.js"></script>
<script src="test2.js"></script>
```

```
// Test1.js
let i = 0;
```

```
// Test2.js
console.log(i);
```

```
0
>
```

  Notice that even though i was declared and initialized in Test1.js, we can still call it using Test2.js.
- Because of this, we may get variable and function naming conflicts and strict mode may be applied to all files.
  Imagine you’re working at a large company and both you and your colleague decide to use the same name for a variable or function. That could cause a lot of confusion and bugs.
- To fix this, we will encapsulate Javascript in a closure.
- **Closure** is a feature in JavaScript where an inner function has access to the outer function’s variables. It allows function/block scopes to be preserved even after they finish executing. In JavaScript, closures are created every time a function is created, at function creation time.
- E.g. Consider the code and output below:

```
function f1(){
    let i = 1;
    function f2(){
        return i;
    }
    return f2;
}

x = f1();
console.log(x());
```

```
1
```

Here I have defined a function within a function. The inner function gains access to all the outer function's local variables, including “i”. The variable “i” is in scope for the inner function.
Normally when a function exits, all its local variables are blown away. However, if we return the inner function and assign it to a variable “x” so that it persists after outer has exited, all of the variables that were in scope when inner was defined also persist.
Note that the variable “i” is totally private to “x”. This is a way of creating private variables in a functional programming language such as JavaScript.
In a language without closure, the variable “i” would have been garbage collected and thrown away when the function outer exited. Calling “x” would have thrown an error because “i” no longer exists.

- To prevent scoping issues, we can “encapsulate” our JavaScript code in a function, like such:

```
(function() {
        "use strict";
        let private = function() {
                // private is not available from outside
        }
}());
```

- We can emulate private variables and methods using closure.
- E.g. Consider the code below:

```
let counter = (function() {
  let privateCounter = 0;

  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment: function() {
      changeBy(1);
    },

    decrement: function() {
      changeBy(-1);
    },

    returnValue: function() {
      return privateCounter;
    }
  };

})();

console.log(counter.returnValue());  // 0.

counter.increment();
counter.increment();
console.log(counter.returnValue());  // 2.

counter.decrement();
console.log(counter.returnValue());  // 1.
```

Here, the variable privateCounter and the function changeBy are private. You can’t access them outside of the anonymous function. However, you can access the 3 public functions, increment, decrement and returnValue from outside the anonymous function.

- Every closure has three scopes:
    1. Local Scope
    2. Outer Functions Scope
    3. Global Scope
- **Note:** When you do something like

```
(function(){
…
})();
```

This is called an **IIFE (Immediately Invoked Function Expression)**.
It is a JavaScript function that runs as soon as it's defined.

HTML5:

- HTML5 has a lot of new features such as:

1. **Geolocation:**

- The HTML Geolocation API is used to get the geographical position of a user.
- Since this can compromise privacy, the position is not available unless the user approves it.
- The **getCurrentPosition()** method is used to return the user's position.
- E.g.

```
navigator.geolocation.getCurrentPosition(success);
function success(position){
        let lat = position.coords.latitude;
        let long = position.coords.longitude;
}
```

2. **Local Storage:**

- The localStorage object stores the data with no expiration date. The data will not be deleted when the browser is closed, and will be available the next day, week, or year.
- **Note:** Local storage is not cookies.
- Local storage stores key/value pairs.
- The storage is bound to the origin (domain/protocol). That is, different protocols or subdomains infer different storage objects, they can't access data from each other.
  I.e. Accessible from the same domain only.
- On Chrome, we can store up to 10 mb in local storage.

3. **Drag and Drop:**

- Can be used for interacting with the DOM or uploading a file.

4. **Canvas:**

- The HTML <canvas> element is used to draw graphics via JavaScript.
- The <canvas> element is only a container for graphics. You must use JavaScript to actually draw the graphics.
- Canvas has several methods for drawing paths, boxes, circles, text, and adding images.

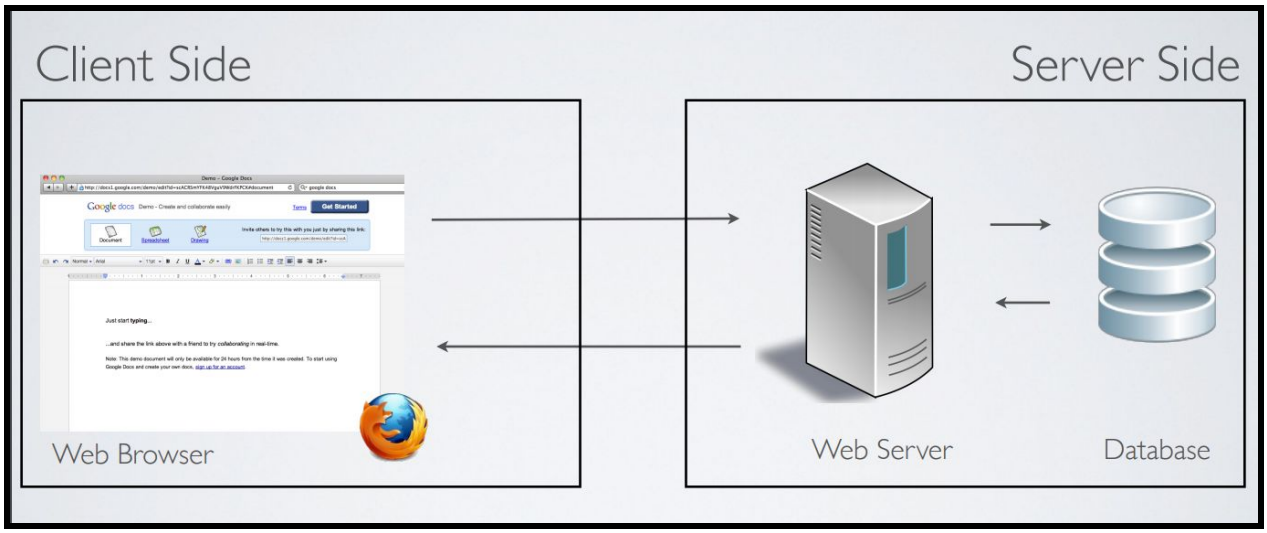
5. **Video:**

- The HTML <video> element is used to show a video on a web page.
- You can embed Youtube videos too.

6. **Speech to Text:**
7. **Text to Speech:**

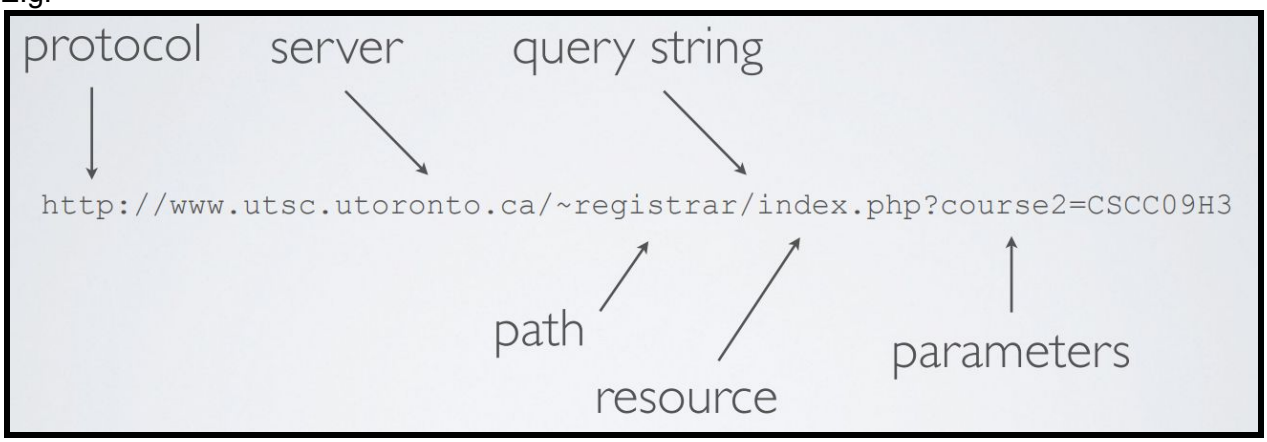
HTTP Protocol:

- **Introduction:**
- HTTP stands for **HyperText Transfer Protocol**.
- The HTTP protocol is a network protocol for requesting/receiving data on the Web.
- A **client** is usually a web browser.
  A **server** is a machine that hosts the website.



- Communication between clients and servers is done by **HTTP requests** and **HTTP responses** like such:
  1. A client sends an HTTP request to the web.
  2. A web server receives the request.
  3. The server runs an application to process the request.
  4. The server returns an HTTP response to the browser.
  5. The client receives the response.
- The server usually runs the standard TCP protocol on port 80 by default.
  I.e. The browser connects to the server on port 80 by default.
  Note: Port 80 is used if we're using HTTP. If we're using HTTPS, the port is 443.
- The **URI (Uniform Resource Identifier)/URL (Uniform Resource Locator)** specifies what resource is being accessed.
- A **Uniform Resource Identifier (URI)** is a unique identifier used by web technologies. URIs may be used to identify anything, including real-world objects, such as people and places, concepts, or information resources such web pages and books. Some URIs provide a means of locating and retrieving information resources on a network (either on the Internet or on another private network, such as a computer filesystem or an Intranet), these are **Uniform Resource Locators (URL)**.
  I.e. A URI is an identifier of a specific resource while a URL is a special type of URI that also tells you how to access it.

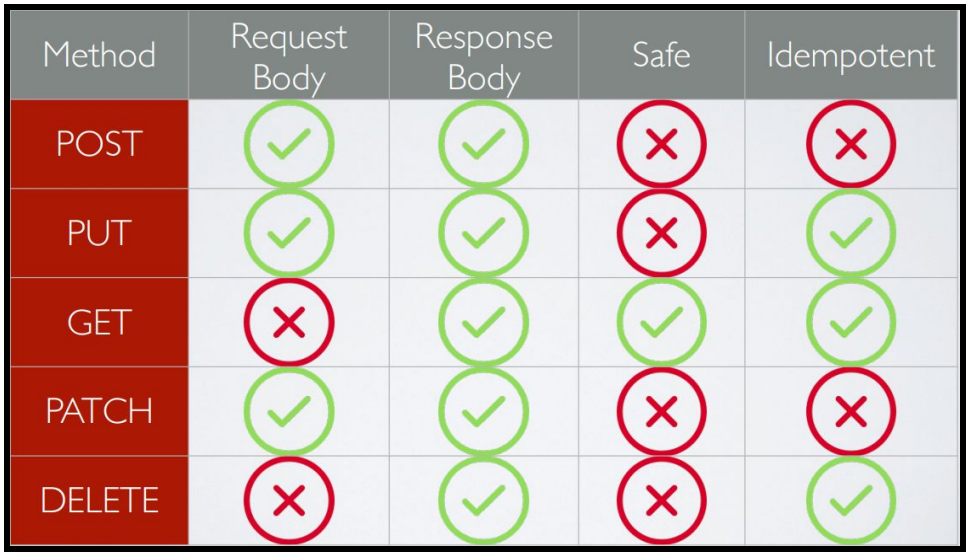
- **Anatomy of a URL:**
- E.g.



- HTTP is the protocol.
- www.utsc.utoronto.ca is the name of the server.
- Then, you have the path which refers to a resource in the webserver.
  Note: Sometimes, you might not see the resource in the URL.
- Then we have a question mark (?). This question mark represents the start of **query parameter(s)**. Query parameters are usually key-value pairs.
  Note: If there are multiple query parameters, they are separated by an ampersand (&).
  E.g. Consider this URL:
  http://www.google.co.uk/search?**q=url&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-GB:official&client=firefox-a**
  Notice that each query parameter is separated by a &.
- **Note:** The query string contains the path, resource and parameters.
- **HTTP Request Methods:**

Request Method	**Explanation**
GET	Requests a representation of the specified resource. Requests using GET should only retrieve data.
POST	Used to submit an entity to the specified resource, often causing a change in state or side effects on the server. I.e. Adds an unidentified resource.
PATCH	Used to update to a resource.
PUT	Adds an identified resource. The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly has side effects of creating the same resource multiple times.
DELETE	Deletes the specified resource.

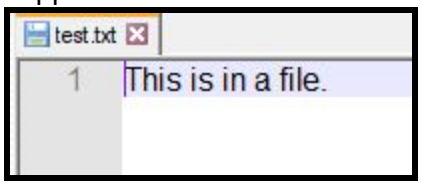
- **HTTP Request:**
- An HTTP request contains the following information:
    - A request method.
    - A query string.
    - A header, which is a key-value pair.
    - An optional body which contains data.
- **Curl Command:**
- We can use the curl command to send HTTP requests to a server.
- Syntax: **curl options url**
  options include:
    - **-v** verbose
    - **--request** request_method
    - **--data** request_body
    - **--header** header
- **HTTP Response:**
- An HTTP response contains the following information:
    - A status code.
    - A header, which is a key-value pair.
    - An optional body which contains data.
- **Status Codes:**
- 1xx - information
- 2xx - success
- 3xx - redirection
- 4xx - client error
  I.e. The client tries to find a web page that doesn't exist.
- 5xx - server errors
- **Method Properties:**
- An HTTP request/response:
    - May have a request body.
    - May have a response body.
    - May not have side effects. **(Safe)**
    - May have the same result when called multiple times. **(Idempotent)**
- The developer/programmer makes the choice.
- What the standard recommends



- **Difference Between HTTP and HTTPS:**
- HTTP is unsecure while HTTPS is secure.
- HTTP sends data over port 80 while HTTPS uses port 443.
- HTTP operates at the application layer, while HTTPS operates at the transport layer.
- No SSL certificates are required for HTTP, but it is required that you have an SSL certificate and it is signed by a CA for HTTPS.
- HTTP doesn't require domain validation, whereas HTTPS requires at least domain validation and certain certificates even require legal document validation.
- There is no encryption in HTTP, whereas with HTTPS the data is encrypted before sending.

JavaScript on the Server:

- We'll be using Node.js for this class.
- Node.js:
  - Runs on Chrome V8 Javascript engine.
  - Has non blocking-IO (**asynchronous/event-driven**).
  - Has no restrictions unlike when js is running on the browser.
- E.g. of a node.js file reading another file:
  Suppose we had a file called "test.txt" with the line "This is in a file." in it.



We can use the following javascript code to read it.

```
const fs = require('fs')
fs.readFile('test.txt', 'utf8', function (err,data) {
  if (err) {
    console.log(err);
  }
  else{
    console.log(data);
  }
});
```

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ node test.js
This is in a file.
```

Note: We couldn't use JavaScript on the browser to read files on a user's computer.

- **Blocking code** is code that runs one instruction after another, and makes next instructions wait.
- **Non-blocking code** means that we don't have to wait for some code to run before running other code.
  I.e. Blocking refers to operations that block further execution until that operation finishes while non-blocking refers to code that doesn't block execution.
- Blocking methods execute synchronously and non-blocking methods execute asynchronously.

- E.g. of Building an HTTP server with Node.js:

```
const http = require('http');
const PORT = 3000;

var handler = function(req, res){
    console.log("Method:", req.method);
    console.log("Url:",req.url);
    console.log("Headers:", req.headers);
    res.end('hello world!');
};

http.createServer(handler).listen(PORT, function (err) {
    if (err) console.log(err);
    else console.log("HTTP server on http://localhost:%s", PORT);
});
```

- We need to process HTTP requests and execute different actions based on:
    - The request method
    - The url path
    - Whether the user is authenticated
- A router can be written from scratch but it is tedious. Instead, we'll use the backend framework Express.js.
  Note: Express is not part of the default library in Node.js. We have to install it.
- E.g. Express.js with multiple request methods:

```
const express = require('express')
const app = express();

// curl localhost:3000/
app.get('/', function (req, res, next) {
    res.end("Hello Get!");
});

// curl -X POST localhost:3000/
app.post('/', function (req, res, next) {
    res.end("Hello Post!");
});

const http = require('http');
const PORT = 3000;

http.createServer(app).listen(PORT, function (err) {
    if (err) console.log(err);
    else console.log("HTTP server on http://localhost:%s", PORT);
});
```

- E.g. Express.js routing based on path:

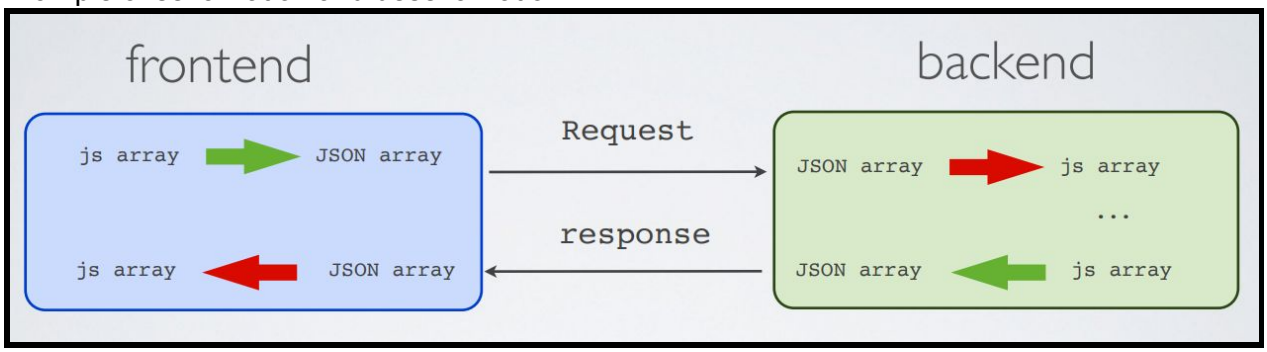
```
// curl localhost:3000/
app.get('/', function (req, res, next) {
    res.end(req.path + ": the root");
});

// curl localhost:3000/messages/
app.get('/messages/', function (req, res, next) {
    res.end(req.path + ": get all messages");
});

// curl localhost:3000/messages/1234/
app.get('/messages/:id/', function (req, res, next) {
    res.end(req.path + ": get the message " + req.params.id);
});
```

- The body of HTTP request and response is a string. A problem arises if we want to pass data structures. 3 possible solutions are encoding it with either:
    1. URI encoding (sometimes used)
    2. XML encoding (rarely used these days)
    3. JSON encoding (very frequently used these days)
- By default, the frontend and backend uses URI encoding. If you want to encode using JSON, you need to tell the frontend/backend that you're using JSON. To do it, you can use "application/json".

JSON:

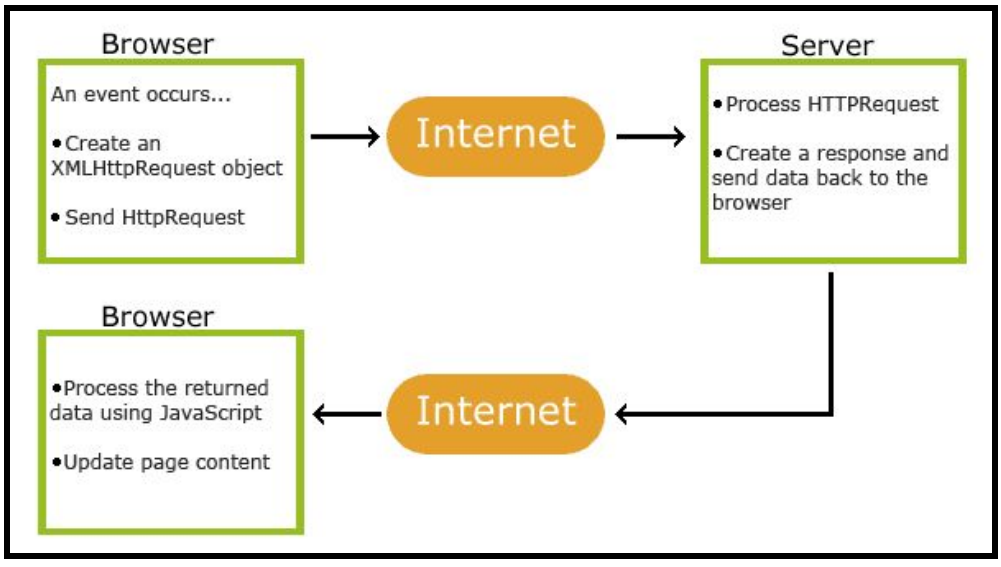
- Stands for **JavaScript Object Notation**.
- Originally, we used URI encoding, but it was hard to send data structures. Then, we used XML, but it's very hard for people to read. Then, someone came up with JSON.
- It is a way to encode data structures into a string. JSON is not a language.
- JSON is a human readable lightweight open format to interchange data.
- Since 2009 browsers support JSON natively.
- A JSON data structure is either:
    1. array (indexed array)
    2. object (associative array)
- JSON values are:
    1. string
    2. number
    3. true
    4. false
    5. null
- **Serialization** means that we're converting JavaScript into JSON.
  I.e. **var myJSONText = JSON.stringify(myObject);**
- **Deserialization** means that we're converting JSON into JavaScript.
  I.e. **var myObject = JSON.parse(myJSONtext);**

- Example of serialization and deserialization:



Ajax:

- AJAX is not a programming language. It is a simple JavaScript command.
- AJAX is a technique for accessing web servers from a web page without refreshing the page.
  I.e. AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.
- AJAX stands for **Asynchronous JavaScript And XML**.
- AJAX revolutionized the web. It started with Gmail and Google Maps.
- Advantages:
  1. Low latency
  2. Rich interactions
- Consequences:
  1. The webapp center of gravity moved to the client side.
  2. Javascript engine performance race.
- AJAX just uses a combination of:
  1. A browser built-in XMLHttpRequest object to request data from a web server.
  2. JavaScript and HTML DOM to display or use the data.
- How AJAX works:



1. An event occurs in a web page.
2. An XMLHttpRequest object is created by JavaScript.
3. The XMLHttpRequest object sends a request to a web server.

4. The server processes the request.
5. The server sends a response back to the web page.
6. The response is read by JavaScript.
7. A proper action, like page update, is performed by JavaScript.

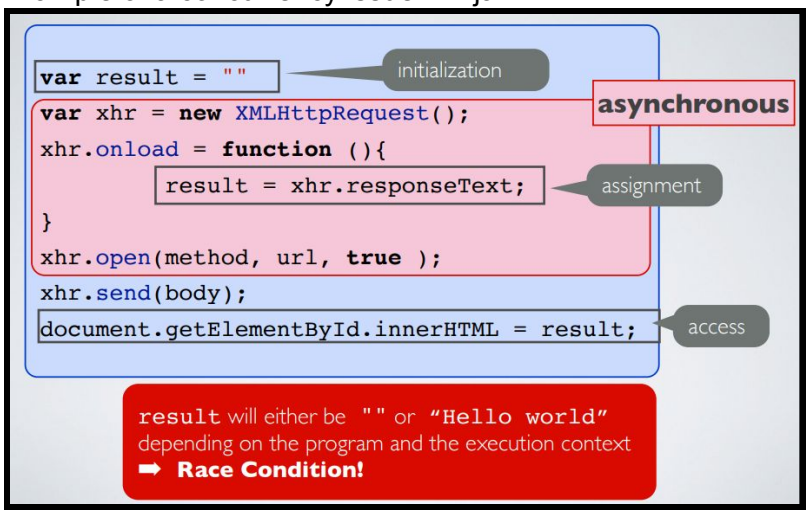
- To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object.
  xhttp.open(method, text, true);
  xhttp.send() or xhttp.send(string);
  Note: Use **xhttp.send()** for GET and **xhttp.send(string)** for POST.
- Server requests should be sent asynchronously. This means that the async parameter of the open() method should be set to true.
- By sending asynchronously, the JavaScript does not have to wait for the server response, but can instead:
  1. Execute other scripts while waiting for server response.
  2. Deal with the response after the response is ready.

  E.g.

```
var xhr = new XMLHttpRequest();
xhr.onload = function(){
  if (xhr.status !== 200)
      console.error("[" + xhr.status + "]" + xhr.responseText);
  else
      console.log(xhr.responseText);
};
xhr.setRequestHeader(key, value);
xhr.open(method, url, true);
xhr.send(body);
```

(always) asynchronous

- Concurrency is a big issue in Ajax.
- Example of a concurrency issue in Ajax.



Callback:

- In JavaScript, functions are objects. Because of this, functions can take other functions as arguments, and can be returned by other functions. Functions that do this are called **higher-order functions**. Any function that is passed as an argument is called a **callback function**.
- Callbacks are a way to make sure certain code doesn't execute until other code has already finished execution.
  I.e. Callbacks make sure that a function is not going to run before a task is completed but will run right after the task has completed. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors.
- The benefit of using a callback function is that you can wait for the result of a previous function call and then execute another function call.
- **Note:** The callback function is helpful when you have to wait for a result that takes time. For example, the data coming from a server because it takes time for data to arrive.
- Since JavaScript is an event driven language, we can use callback functions for event declarations, too.
- Example that JavaScript is asynchronous.
  Consider this Python program:

```python
import time

def first():
    time.sleep(10) # Sleeps for 10 seconds.
    print("First")

def second():
    print("Second")

first()
second()
```

When you run it, it waits for 10 seconds before printing First and then prints Second.
I.e. It runs in a sequential order.

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop/test2$ python3 test.py
First
Second
```

Consider this JavaScript program that does the same thing:

```javascript
function first(){
  // Simulate a code delay for ten seconds
  setTimeout( function(){
    console.log(1);
  }, 1000);
}
function second(){
  console.log(2);
}
first();
second();
```

When you run the program, even though you call first() first, 2 will be printed first.

This is because JavaScript is asynchronous.
Note: In the above JavaScript file, the function in setTimeout is an example of a callback function.

- Example of callback functions used in events.

Consider the HTML and JavaScript code below:

```
<!DOCTYPE html>
<html>
<head>
    <script src="test.js"></script>
</head>
<body>
    <button id="button"> Click Me </button>
</body>
</html>
```

```
(function(){
    window.addEventListener("load", function(){

        document.querySelector("#button").addEventListener("click", function() {
            console.log("User has clicked on the button!");
        });

    });
}());
```

We have 2 callback functions in the JavaScript code.
The first is in window.addEventListener and the second is in document.querySelector("#button").addEventListener.
The functions won't run unless the event is triggered.

REST APIs:

- An **API** is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information user.
  I.e. If you want to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you want to that system so it can understand and fulfill the request.
  I.e. An API is a set of rules that allows programs to talk to each other.
- An **endpoint** is the location from which APIs can access the resources they need to carry out their function. For web APIs, endpoints are usually URLs.
- **REST** stands for **representational state transfer** and was created by computer scientist Roy Fielding.
- **REST APIs** are also known as **RESTful APIs**.
- A REST API is an API that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services.
  Note: REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways.
- When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This representation is delivered in one of several formats via HTTP, of which JSON is the most popular to use because it's language-agnostic and readable by both humans and machines.
- In order for an API to be considered RESTful, it has to conform to these criteria:
  1. A client-server architecture made up of clients, servers, and resources, with requests managed through HTTP.
  2. Stateless client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected.
  3. Cacheable data that streamlines client-server interactions.
  4. A uniform interface between components so that information is transferred in a standard form. This requires that:
     a. resources requested are identifiable and separate from the representations sent to the client.
     b. resources can be manipulated by the client via the representation they receive because the representation contains enough information to do so.
     c. self-descriptive messages returned to the client have enough information to describe how the client should process it.
     d. hypertext/hypermedia is available, meaning that after accessing a resource the client should be able to use hyperlinks to find all other currently available actions they can take.
  5. A layered system that organizes each type of server involved in the retrieval of requested information into hierarchies, invisible to the client.
  6. Code-on-demand: The ability to send executable code from the server to the client when requested, extending client functionality.
- The function names of a REST API consist of the HTTP method and the URL.
- The function arguments of a REST API consist of a URL and the request body. The return value of a function is a status code and the response body.

- E.g.

	HTTP request	HTTP response
Create a new message	`POST /messages/ "Hello World"`	`200 "78"`
Get all messages	`GET /messages/`	`200 "['Hello world', ...]"`
Get a specific messages	`GET /messages/78/`	`200 "Hello World"`
Delete a specific messages	`DELETE /messages/78/`	`200 "success"`

- The server is more or less a storage system that stores:
    1. Collections/Resources
    2. Elements that belong to one or several collections.

  For example, in the example above, in the third row, messages is a collection while 78 is an element.
- Usually, the pattern is collection/element/collection/element/etc.
- E.g.

Type	Example
one-to-one	`/users/sansthie/profile/firstname/`
one-to-many	`/users/sansthie/messages/89/`
many-to-many	`/users/sansthie/teams/8/` `/teams/8/users/sansthie/`

  Here, in the first row, users is a collection while sansthie is an element, and profile is a collection while firstname is an element.
- REST APIs have 3 relationships:
    1. One-to-one
    2. One-to-many
    3. Many-to-many

CRUD Operations:

- CRUD stands for:
  - Create
  - Read
  - Update
  - Delete
- They are 4 basic functions of persistent storage.
- How HTTP methods map to CRUD operations:

CRUD	HTTP	Collection	Element
Create	POST		Create a new element
	PUT	Replace the entire collection	Create (or replace if exists) a specific element
Read	GET	List all elements	Retrieve a specific element
Update	PATCH	Update some attributes of some elements	Update some attributes of a specific element
Delete	DELETE	Delete the entire collection	Delete a specific element

- **PUT vs POST:**

PUT	**POST**
PUT is idempotent. So if you send a request multiple times, that should be equivalent to a single request modification.	POST is NOT idempotent. So if you send a request N times, you will end up having N resources.
Use PUT when you want to modify a singular resource which is already a part of resources collection. PUT replaces the resource in its entirety. Use PATCH if the request updates part of the resource.	Use POST when you want to add a child resource under resources collection.
Generally, in practice, use PUT for UPDATE operations that replaces the resource in its entirety.	Use POST for CREATE operations.

Note: These are just guidelines. The implementation details are left up to the developers.

- We can use attributes to query/filter a subset of a collection.
  E.g. GET /messages/?from=67&to=99

- **HTTP Methods for RESTful Services:**

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists.
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

Note: Once again, these are just the guidelines.

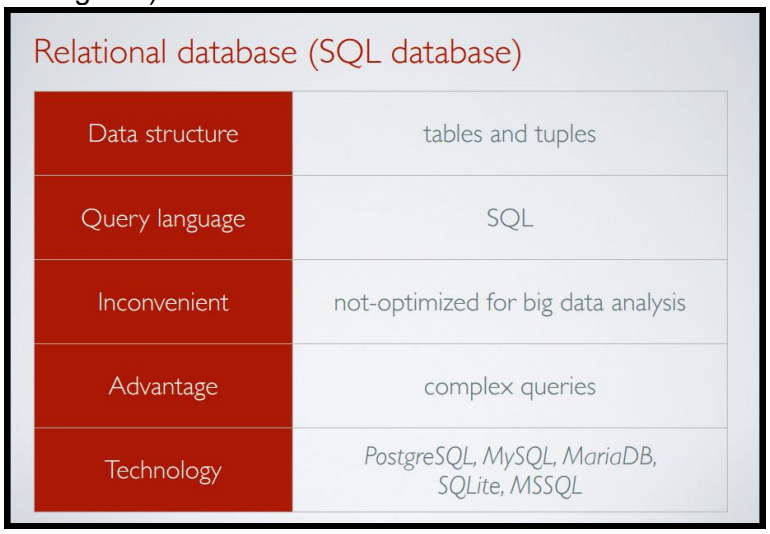
Handling Data:

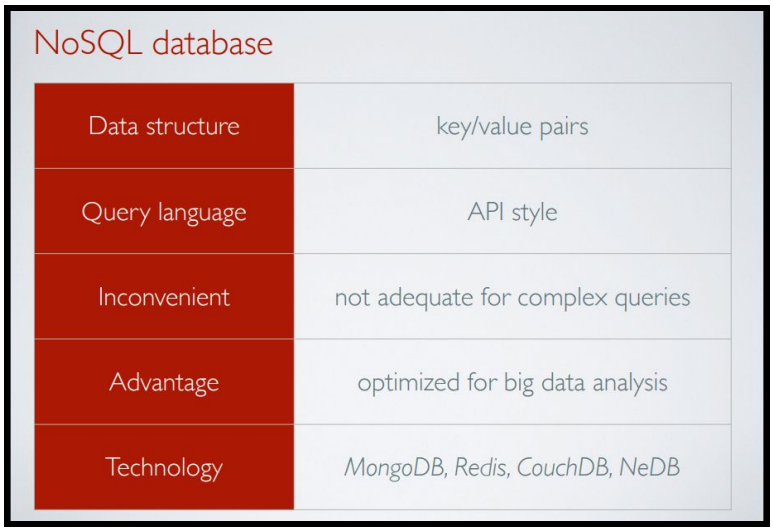
- To handle data, we'll use a database.
- We use a database because:
  - Persistency
  - Concurrency (avoid race conditions)
  - Query
  - Scalability
- **SQL vs NOSQL databases:**

Parameter	SQL	NOSQL
Type	Are table based databases	Can be document based, key-value pairs, or graph databases.
Schema	Have a predefined schema	Use dynamic schema for unstructured data.
Ability to scale	Are vertically scalable	Are horizontally scalable

- The concept of NoSQL databases became popular with internet giants like Google, Facebook, Amazon, etc who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could **scale up** our systems by upgrading our existing hardware. This process is expensive. (SQL uses scaling up.)
The alternative for this issue is to distribute the database load on multiple hosts whenever the load increases. This method is known as **scaling out**. (NOSQL uses scaling out.)





- **ORM (Object Relational Mapping):**
- **ORM** is a technique that lets you query and manipulate data from a database using an object-oriented paradigm. It provides a mapping between objects and the database structure.
- An ORM library is a library written in your language of choice that encapsulates the code needed to manipulate the data, so you don't use SQL anymore. You interact directly with an object in the same language you're using.
- A few advantages of using ORMs are:
    - You write your data model in only one place, and it's easier to update, maintain, and reuse the code. (DRY principle)
    - It sanitizes the query statement, preventing SQL injections.
- Examples:
    - Sequelize for PostgreSQL, MySQL, MariaDB, SQLite
    - Mongoose for MongoDB

Connecting the REST API with a database:

- Do retrieve selected elements only rather than retrieving an entire collection and filtering afterwards. This is because the database is going to be bigger than the memory capacity of the backend, so if you retrieve an entire collection, it could be dangerous.
- Do define primary keys rather than relying on auto-generated ones.
- Do split data into different collections rather than storing list attributes.
- Do create join collections whenever appropriate (only for NoSQL databases without performant join feature).
- Only retrieve what you need from a potentially large collection. (Use pagination)

Handling Files:

- Recall that JavaScript in the browser cannot open and read files on the user's computer/desktop.
- The only way to upload files is through file input forms.
  I.e.

```
<form . . . >
    <input type="file" name="img" multiple>
    (The multiple lets users upload multiple files at once.)
    <input type="submit">
</form>
```

- There are 2 ways to send a file from the browser:
  1. Old Way:
  - Form action (with page refresh)
    E.g.

```
<form action="/url"
        method="POST"
        enctype="multipart/form-data">
```

  - **Note:** The enctype="multipart/form-data" tells the server that we're going to send a form that contains both text and binary data.
  2. New Way:
  - Ajax request (without page refresh)
    E.g.

```
var file = document.get ...
var formdata = new FormData();
formdata.append("picture", file);
xhr.send(formdata);
```

  - **Note:** We use FormData because we're sending a mix of binary and text.
- The best approach is to store files on discs, but you can store files in databases.
- The server receives the following information:
  - File metadata, which includes the filename, mimetype (file type), size and others.
  - File content which is a compressed binary or string.
- **MIME Type:**
- **MIME (Multipurpose Internet Mail Extensions)** is also known as the content type.
- It defines the format of a document exchanged on the internet.
- **Do/Don't with files:**
- Do not send a base64 encoded file content with JSON, use multipart/form-data instead (compression).
- Do not store uploaded files with the static content.
- Do not serve uploaded files statically. If you upload files statically, everyone can access it, causing security issues.
- Do store the mimetype and set the HTTP response header mimetype when files are sent back.

Security:

- You have absolutely no control on the client.
- Users can modify the frontend code.

Cookies:

- **Introduction to cookies:**
- **Cookies** are text files with small pieces of key-value pairs of data that are used to identify your computer as you use a computer network. Data stored in a cookie is created by the server upon your connection. This data is labeled with an ID unique to you and your computer. When the cookie is exchanged between your computer and the network server, the server reads the ID and knows what information to specifically serve to you.
- Cookies are embedded in the headers of HTTP requests and responses.
  I.e. Cookies are key/value pairs sent back and forth between the browser and the server in HTTP request and response.
- Cookies:
  - Contain text data (Up to 4kb)
  - May or may not have an expiration date
  - Are bound to a domain name and a path.
    I.e. Every website has cookies and your browser links/connects that website with those cookies.
  - May have security flags
  - Can be manipulated from the client and the server
- Cookies are good for:
  - Shopping cart
  - Browsing preferences
  - User authentication
  - Tracking and advertisement
- The purpose of the computer cookie is to help the website keep track of your visits and activity. For example, many online retailers use cookies to keep track of the items in a user's shopping cart as they explore the site. Without cookies, your shopping cart would reset to zero every time you clicked a new link on the site.
- A website might also use cookies to keep a record of your most recent visit or to record your login information. Many people find this useful so that they can store passwords on frequently used sites, or simply so they know what they have visited or downloaded in the past.
- There are 2 main types of cookies:
  1. **Magic Cookies:**
  - Magic cookies are an old computing term that refers to packets of information that are sent and received without changes. This concept predates the modern cookie we use today.
  2. **HTTP Cookies:**
  - HTTP cookies are a repurposed version of the magic cookie built for internet browsing. Web browser programmer Lou Montulli used the magic cookie as inspiration in 1994. He recreated this concept for browsers when he helped an online shopping store fix their overloaded servers.
  - HTTP cookies, or internet cookies, are built specifically for internet web browsers to track, personalize, and save information about each user's session.
  - Cookies are created to identify you when you visit a new website. The web server sends a short stream of identifying info to your web browser.
  - Browser cookies are identified and read by name-value pairs. These tell cookies where to be sent and what data to recall.

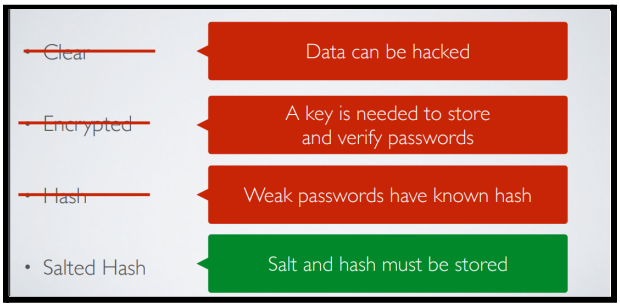
- Websites use HTTP cookies to streamline your web experiences. Without cookies, you'd have to login again after you leave a site or rebuild your shopping cart if you accidentally close the page.
- **Here's how HTTP cookies are intended to be used:**
    1. **Session management:** For example, cookies let websites recognize users and recall their individual login information and preferences.
    2. **Personalization:** Customized advertising is the main way cookies are used to personalize your sessions. You may view certain items or parts of a site, and cookies use this data to help build targeted ads that you might enjoy.
    3. **Tracking:** Shopping sites use cookies to track items users previously viewed, allowing the sites to suggest other goods they might like and keep items in shopping carts while they continue shopping.
- **Different types of HTTP cookies:**
- There are 2 main types of cookies:
    1. **Session cookies** are used only while navigating a website. They are stored in random access memory and are never written to the hard drive.
       When the session ends, session cookies are automatically deleted. They also help the "back" button or third-party anonymizer plugins work. These plugins are designed for specific browsers to work and help maintain user privacy.
    2. **Persistent cookies** remain on a computer indefinitely, although many include an expiration date and are automatically removed when that date is reached.
       Persistent cookies are used primarily for authentication and tracking.
       For authentication, cookies track whether a user is logged in and under what name. They also streamline login information, so users don't have to remember site passwords.
       For tracking, cookies track multiple visits to the same site over time. For example, some online merchants use cookies to track visits from particular users, including the pages and products viewed. The information they gain allows them to suggest other items that might interest visitors. Gradually, a profile is built based on a user's browsing history on that site.
- **Why cookies can be dangerous:**
- While cookies can't infect computers with viruses or other malware, the danger lies in their ability to track individuals' browsing histories.
- **First-party cookies** are directly created by the website you are using. These are generally safe, as long as you are browsing reputable websites or ones that have not been compromised.
- **Third-party cookies** are more troubling. They are generated by websites that are different from the web pages users are currently surfing, usually because they're linked to ads on that page. Visiting a site with 10 ads may generate 10 cookies, even if users never click on those ads. Third-party cookies let advertisers or analytics companies track an individual's browsing history across the web on any sites that contain their ads.
- **Zombie cookies** are from a third-party and are permanently installed on the users' computers, even when they opt not to install cookies. They also reappear after they've been deleted. Like other third-party cookies, zombie cookies can be used by web analytics companies to track unique individuals' browsing histories. Websites may also use zombies to ban specific users.
- **Manipulating cookies:**
- A cookie can be modified, as long as there is no cookie flag set.
- On the server side, we can use cookie in Express.
  E.g. **const cookie = require('cookie');** ← Used in Express.
- On the client side, we can use Document.cookie in Javascript.

Sessions:

- **Introduction to sessions:**
- A **web session** is a series of contiguous actions by a visitor on an individual website within a given time frame. This could include your search engine searches, filling out a form to receive content, scrolling on a website page, adding items to a shopping cart, researching airfare, or which pages you viewed on a single website. Any interaction that you have with a single website is recorded as a web session to that website property.
- To track sessions, a **web session ID** is stored in a visitor's browser. This session ID is passed along with any HTTP requests that the visitor makes while on the site.
- To avoid storing massive amounts of information in-browser, developers use session IDs to store information server-side while enabling user privacy. Every time a user makes an action or makes a request on a web application, the application sends the session ID and cookie ID back to the server, along with a description of the action itself.
- Once a web developer accrues enough information on how users traverse their site, they can start to create very personalized, engaging experiences.
- A session can be defined as a server-side storage of information that is desired to persist throughout the user's interaction with the web site or web application.
- Instead of storing large and constantly changing information via cookies in the user's browser, only a unique identifier is stored on the client side called a session id. This session id is passed to the web server every time the browser makes an HTTP request. The web application pairs this session id with it's internal database and retrieves the stored variables for use by the requested page.
- **General concepts of sessions:**
- There is a session id, a token, between the browser and the web application.
- The session id should be unique and unforgeable. It is usually a long random number or a hash.
- The session id is stored in a cookie while session key/value pairs are stored on the server.
- The user can create, modify, delete the session ID in the cookie, but cannot access the key/value pairs stored on the server.

Web Authentication:

- There are several ways to do web authentication:
    1. **Local authentication:**
    - Manage username and password yourself.
      I.e. A user creates an account. You store the username and password in a database, and each time a user wants to log in, you compare the inputted username and password to the ones you have in the database.
    - **How to store passwords:**



    - **Salted hash** means that we're going to add a random string (the salt) to the password to make it strong. Each new salted password will be unique as the salt is always unique. Then, we'll hash the salted password. Salted hash is resistant to brute force attacks.

- **Basic/Stateless Authentication:**
- (Standard) RFC 2617
- Login and password are sent in clear (Base64 encoding) in the headers "authorization".
- Pros:
    - Since there aren't many operations going on, authentication can be faster with this method.
    - Easy to implement.
    - Supported by all major browsers.
- Cons:
    - Base64 is not the same as encryption. It's just another way to represent data. The base64 encoded string can easily be decoded since it's sent in plain text. This poor security feature calls for many types of attacks. Because of this, HTTPS/SSL is absolutely essential.
    - Credentials must be sent with every request.
    - Users can only be logged out by rewriting the credentials with an invalid one.
- **Session/Stateful Authentication:**
- (Standard) RFC 6265
- Uses cookies.
- The user enters a login and password and the frontend sends them to the backend (POST request).
  Then, the backend verifies the login/password based on information stored on the server (usually in the database).
  Then, the backend stores user information in a session.
  Then, the backend grants access to resources based on the information contained in the session.
- Pros:
    - Faster subsequent logins, as the credentials are not required.
    - Improved user experience.
    - Fairly easy to implement. Many frameworks provide this feature out-of-the-box.
- Cons:
    - Cookies are sent with every request, even if it does not require authentication
- **Do/Don't with passwords:**
- On the client side, either send passwords in the headers (automatic with basic authentication) or in the body (POST request with session authentication). Never send/show passwords in the URL.
- On the server, store passwords as salted hash passwords only. Never store passwords in clear or non-salted hash.

2. Token-based authentication:

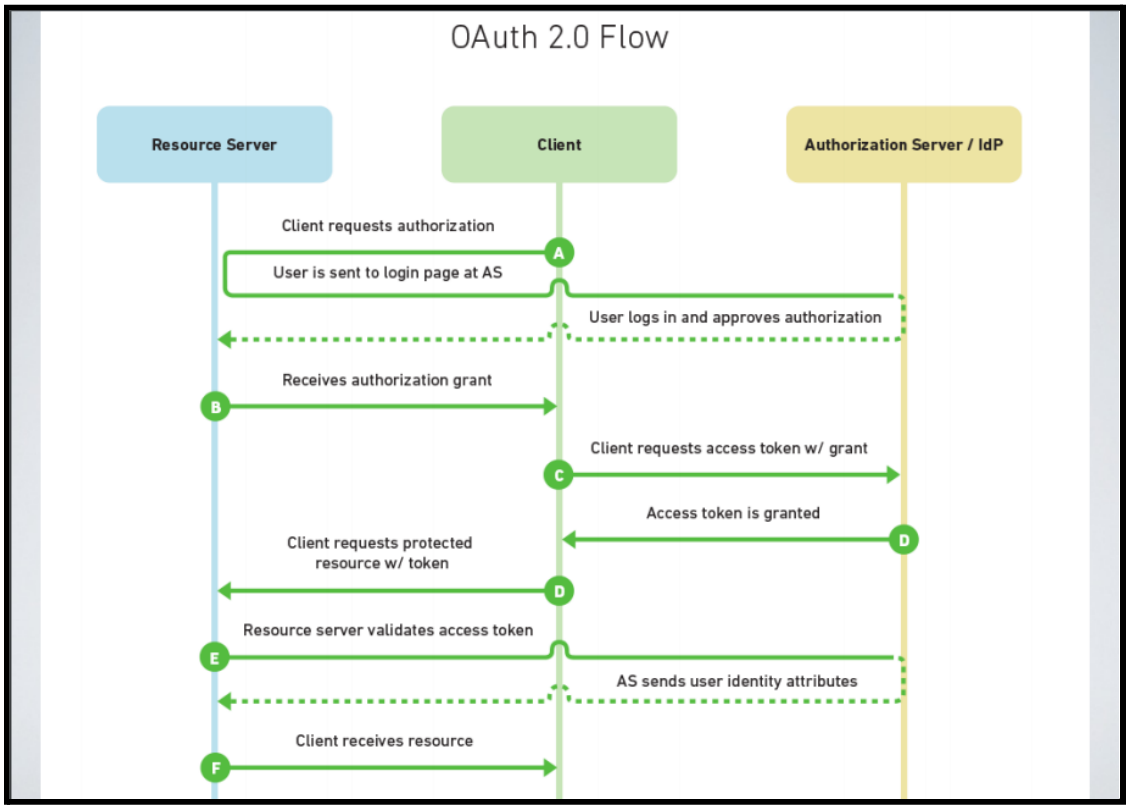
- This method uses tokens to authenticate users instead of cookies. The user authenticates using valid credentials and the server returns a signed token. This token can be used for subsequent requests.
- **HMAC:**
- (Standard) RFC 2104
- For each authenticated HTTP request, the frontend computes and sends a message digest that combines the user's secret and some request arguments.
- The user's password never transits back and forth except perhaps for the first time it is exchanged.

- The digest can be sent in clear. Should not store sensitive information in the digest.
- **JSON Web Token:**
- (Standard) RFC 7519
- Encodes the user's information in a string (token) that is URL safe.
- The token is usually authenticated and sometimes encrypted.
- The web token can be used for stateful but yet session-less authentication.
- Stateless JSON Web Token is a self-contained token which does not need any representation on the backend.
- Stateful JSON Web Token is a token which contains only part of the required data such as session/user ID and the rest is stored on the server side.
- Revoking tokens can be complicated.

3. **Third party authentication:**

- You can sign into a website through signing into another website.
- **Single sign-on (SSO)** is an authentication scheme that allows a user to log in with a single ID and password to any of several related, yet independent, software systems.
- There are many types of SSO, such as Pubcookie, OpenID, SAML, OAuth, among others.
- **Social login** is a form of single sign-on using existing information from a social networking service such as Facebook, Twitter or Google, to sign into a third party website instead of creating a new login account specifically for that website. It is designed to simplify logins for end users as well as provide more and more reliable demographic information.
- **OpenID:**
- An HTTP based protocol that uses an identity provider to validate a user. The user's password is secured with one identity provider. This allows other service providers a way to achieve SSO without requiring a password from the user.
- **SAML:**
- Is XML based.
- Is used in many enterprise applications to enable enterprises to monitor who has access to corporate resources.
- **OAuth 2.0:**
- (Standard) RFC 6749
- Is JSON based.
- OAuth 2.0 is a security standard where you give one application permission to access your data in another application. You authorize one application to access your data, or use features in another application on your behalf, without giving them your password. OAuth 2.0 does this by allowing a token to be issued by the identity provider to these third party applications, with the approval of the user.
- OAuth doesn't share password data but instead uses authorization tokens to prove an identity between consumers and service providers. OAuth is an authentication protocol that allows you to approve one application interacting with another on your behalf without giving away your password.
- How it works:
    1. The backend redirects the user to the third-party login-page.
    2. The third-party asks and verifies the login/password based on the third party user information.
    3. The third party redirects the user back to the application with an OAuth token and verifier in the url.
    4. Backend verifies the token with the third party.

5. Backend starts a session.



- The user's login/password never transit by the application’s frontend or backend.

Securing the web architecture means securing:

- The network
- The operating system
- The web server (Apache for instance)
- The administration server (SSH for instance)
- The database (Oracle for instance)
- The web application ← Our focus

Insufficient Transport Layer Protection a.k.a the need for HTTPs:

- While hackers can try to brute force a user's password/session ID, a more common and better method is to steal the user's password or session ID. Brute forcing usually doesn't work.
- Hackers can eavesdrop and/or tamper with the messages sent back and forth between your browser and the server. This is known as **man in the middle attack (MitM) attack**.
- MitM attacks consist of sitting between the connection of two parties and either observing or manipulating traffic. This could be through interfering with legitimate networks or creating fake networks that the attacker controls. Compromised traffic is then stripped of any encryption in order to steal, change or reroute that traffic to the attacker's destination of choice (such as a phishing log-in site). Because attackers may be silently observing or re-encrypting intercepted traffic to its intended source once recorded or edited, it can be a difficult attack to spot.
- A generic solution we can use is to use HTTPS over HTTP.
- Because HTTP was originally designed as a clear text protocol, it is vulnerable to man in the middle attacks. By including SSL/TLS encryption, HTTPS prevents data sent over the internet from being intercepted and read by a third party. Through public-key cryptography and the SSL/TLS handshake, an encrypted communication session can be securely set up between two parties via the creation of a shared secret key.
- Hypertext transfer protocol secure (HTTPS) is the secure version of HTTP, which is the primary protocol used to send data between a web browser and a website. HTTPS is encrypted in order to increase security of data transfer. This is particularly important when users transmit sensitive data, such as by logging into a bank account, email service, or health insurance provider.
- HTTPS = HTTP + TLS
- HTTPS uses an encryption protocol to encrypt communications. This protocol is called Transport Layer Security (TLS), although formerly it was known as Secure Sockets Layer (SSL). TLS secures communications by using an asymmetric public key infrastructure. This type of security system uses two different keys to encrypt communications between two parties:
    1. The private key: This key is controlled by the owner of a website and it's kept, as the reader may have speculated, private. This key lives on a web server and is used to decrypt information encrypted by the public key.
    2. The public key: This key is available to everyone who wants to interact with the server in a way that's secure. Information that's encrypted by the public key can only be decrypted by the private key.
- HTTPS includes robust authentication via the SSL/TLS protocol. A website's SSL/TLS certificate includes a public key that a web browser can use to confirm that documents sent by the server have been digitally signed by someone in possession of the corresponding private key. If the server's certificate has been signed by a publicly trusted **certificate authority (CA)**, the browser will accept that any identifying information included in the certificate has been validated by a trusted third party.
- **Note:** Self-signed certificates are not trusted by your browser.
- Your browser trusts many CAs by default.

- Transport Layer Security provides:
    1. confidentiality: end-to-end secure channel
    2. integrity: authentication handshake
- HTTPS prevents websites from having their information broadcast in a way that's easily viewed by anyone snooping on the network. When information is sent over regular HTTP, the information is broken into packets of data that can be easily "sniffed" using free software. This makes communication over an unsecure medium, such as public Wi-Fi, highly vulnerable to interception. In fact, all communications that occur over HTTP occur in plain text, making them highly accessible to anyone with the correct tools, and vulnerable to on-path attacks. With HTTPS, traffic is encrypted such that even if the packets are sniffed or otherwise intercepted, they will come across as nonsensical characters.
- HTTPS protects any data send back and forth including:
    - login and password
    - session ID
- HTTPS must be used during the entire session. This is because of **mixed-content attacks**. **Mixed-content** happens when an HTTPS page contains elements such as ajax, js, image, video, css, etc that is served with HTTP and an HTTPS page transfers control to another HTTP page within the same domain. Then, the authentication cookie will be sent over HTTP.
- In addition, we can create cookies with the secure flag. A cookie with the Secure attribute is sent to the server only with an encrypted request over the HTTPS protocol, and therefore can't easily be accessed by a man-in-the-middle attacker. Insecure sites, with http: in the URL, can't set cookies with the Secure attribute. However, do not assume that Secure prevents all access to sensitive information in cookies.
  I.e. The Secure attribute makes it so that the cookie will be sent over HTTPS exclusively and will prevent authentication cookies from leaking in case of mixed-content.
- Do/Don't with HTTPS:
    - Always use HTTPS exclusively in production.
    - Always have a valid and signed certificate (no self-signed cert).
    - Always avoid using absolute URL (mixed-content).
    - Always use a secure cookie flag with an authentication cookie.
- **Note:** HTTPS protects against man in the middle attacks but can't protect against hackers who are in your browser or are on the server.
- Other types of vulnerabilities:

Frontend Vulnerabilities	**Backend Vulnerabilities**
Content Spoofing	Incomplete mediation
Cross-Site Scripting	Information leakage
Cross-site Request forgery	SQL injection

Incomplete Mediation:

- Occurs when the application accepts bad/invalid/malicious data from the frontend and that data causes issues.
  I.e. It occurs when failure to perform "sanity checks" on data can lead to random or carefully planned flaws.
- Data coming from the frontend cannot be trusted.
- Sensitive operations must be done on the backend.

Information Leakage:

- **Information leakage** happens whenever a system that is designed to be closed to an eavesdropper reveals some information to unauthorized parties nonetheless.
  I.e. Information leakage occurs when secret information correlates with, or can be correlated with, observable information.
- In its most common form, information leakage is the result of one or more of the following conditions:
  1. A failure to scrub out HTML/script comments containing sensitive information.
  2. Improper application or server configurations.
  3. Differences in page responses for valid vs. invalid data.
- Sensitive information may be present within HTML comments, error messages, source code, or left in plain sight, and there are many ways a website can be coaxed into revealing this type of information. While information leakage doesn't necessarily represent a security breach, it gives an attacker useful guidance for future exploitation.
- A solution to information leakage is to use authentication (I.e. Who are the authorized users?) and to use authorization (I.e. Who can access what and how?).

SQL Injection:

- **SQL injection** is a type of an injection attack that makes it possible to execute malicious SQL statements. These statements control a database server behind a web application. Attackers can use SQL injection vulnerabilities to bypass application security measures. They can go around authentication and authorization of a web page or web application and retrieve the content of the entire SQL/NoSQL database. They can also use SQL injection to add, modify, and delete records in the database.
- An SQL injection usually occurs when you ask a user for input and the user gives you an SQL statement that you will unknowingly run on your database.

Content Spoofing:

- **Content spoofing** allows the end user of the vulnerable web application to spoof or modify the actual content on the web page. The user might use the security loopholes in the website to inject the content that he/she wishes to the target website. When an application does not properly handle user supplied data, an attacker can supply content to a web application, typically via a parameter value, that is reflected back to the user.
- An attacker can inject HTML tags in the page. They will add illegitimate content to the webpage (ads most of the time).
- A generic solution is to validate data inserted in the DOM.

Cross-Site Scripting (XSS):

- **Cross-Site Scripting (XSS) attacks** target scripts embedded in a page that are executed on the client-side rather than on the server-side. Cross-site scripting is one of the most common application-layer web attacks. XSS in itself is a threat that is brought about by the internet security weaknesses of client-side scripting languages, such as HTML and JavaScript. The concept of XSS is to manipulate client-side scripts of a web application to execute in the manner desired by the malicious user. Such a manipulation can embed a script in a page that can be executed every time the page is loaded, or whenever an associated event is performed.
- Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.
- An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute

the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.

- An attacker can inject arbitrary javascript code in the page that will be executed by the browser. Then, the attacker can:
    - Inject illegitimate content in the page (same as content spoofing)
    - Perform illegitimate HTTP requests through Ajax (same as a CSRF attack)
    - Steal Session ID from the cookie
    - Steal user's login/password by modifying the page to forge a perfect scam
- Some variations on XSS attacks are:
    1. **Reflected XSS:** Malicious data sent to the backend are immediately sent back to the frontend to be inserted into the DOM.
    2. **Stored XSS:** Malicious data sent to the backend are stored in the database and later-on sent back to the frontend to be inserted into the DOM.
    3. **DOM-based attack:** Malicious data are manipulated in the frontend (javascript) and inserted into the DOM.
- A generic solution is to validate data inserted in the DOM.
- Another solution is to use the HttpOnly cookie flag. This makes it so that the cookie is not readable/writable from the frontend. This prevents the authentication cookie from being leaked when an XSS attack occurs.
  Note: The name is a little misleading. HttpOnly has nothing to do with HTTP or HTTPS.

Cross-Site Request Forgery (CSRF):

- The **same origin policy** is a critical security mechanism that restricts how a document or script loaded from one origin can interact with a resource from another origin. It helps isolate potentially malicious documents, reducing possible attack vectors. Two URLs have the same **origin** if the protocol, port, and host are the same for both.
  This means https://api.mydomain.com and https://mydomain.com are actually different origins and thus impacted by same-origin policy. In a similar way, http://localhost:9000 and http://localhost:8080 are also different origins.
- **Note:** The path or query parameters are ignored when considering the origin.
- **Note:** Internet Explorer has an exception to the definition of origin. IE treats all ports the same way. This is non-standard and no other browser behaves this way.
- Elements under control of the same-origin policy include:
    - Ajax requests
    - Form actions
- Elements not under control of the same-origin policy include:
    - Javascript scripts
    - CSS
    - Images, video, sound
    - Plugins
- **Cross-Site Request Forgery (CSRF)** is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. CSRFs are typically conducted using malicious social engineering, such as an email or link that tricks the victim into sending a forged request to a server. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

- Consider this example:
  Suppose you log into https://bank.com and a cookie is stored. While logged in, suppose you unknowingly browse a malicious website. Without the same origin policy, the malicious website could make authenticated malicious AJAX calls to https://bank.com/api to POST /withdraw even though the hacker website doesn't have direct access to the bank's cookies.

  This is because many websites use cookies to keep track of authentication or session info. Those cookies are bound to a certain domain when they are created. On every HTTP call to that domain, the browser will attach the cookies that were created for that domain. Furthermore,the browser automatically attaches any cookies bound to https://bank.com for any HTTP calls to that domain, including AJAX calls from the malicious website. By restricting HTTP calls to only ones to the same origin, the same-origin policy closes some hacker backdoors such as around CSRF attacks.
- **Cross-origin resource sharing (CORS)** is a security mechanism that allows a web page from one origin to access a resource with a different domain. CORS is a relaxation of the same-origin policy implemented in modern browsers. Without features like CORS, websites are restricted to accessing resources from the same origin through what is known as same-origin policy.
- There are legitimate reasons for a website to make cross-origin HTTP requests.
  A single-page app at https://mydomain.com could need to make AJAX calls to https://api.mydomain.com.
  Furthermore, some websites, such as Reddit, allow users to embed images from other websites, like Imgur.
- There are 2 types of CORS requests:
  1. **Preflighted Requests:**
     - For Ajax and HTTP request methods that can modify data, the specification mandates that browsers preflight the request, solicit supported methods from the server with an HTTP OPTIONS request method, and then, upon approval from the server, send the actual request with the actual HTTP request method.
       I.e. When performing certain types of cross-domain Ajax requests, modern browsers that support CORS will initiate an extra "preflight" request to determine whether they have permission to perform the action. Cross-origin requests are preflighted this way because they may have implications to user data.
     - A **preflighted request** is a CORS request where the browser is required to send a preflight request (a preliminary check) before sending the request being preflighted to ask the server permission if the original CORS request can proceed.
     - E.g.
       This is the preflight request:
       OPTIONS /
       Host: service.example.com
       Origin: http://www.example.com
       Access-Control-Request-Method: PUT

       If service.example.com is willing to accept the action, it may respond with the following headers:

       Access-Control-Allow-Origin: http://www.example.com
       Access-Control-Allow-Methods: PUT, DELETE

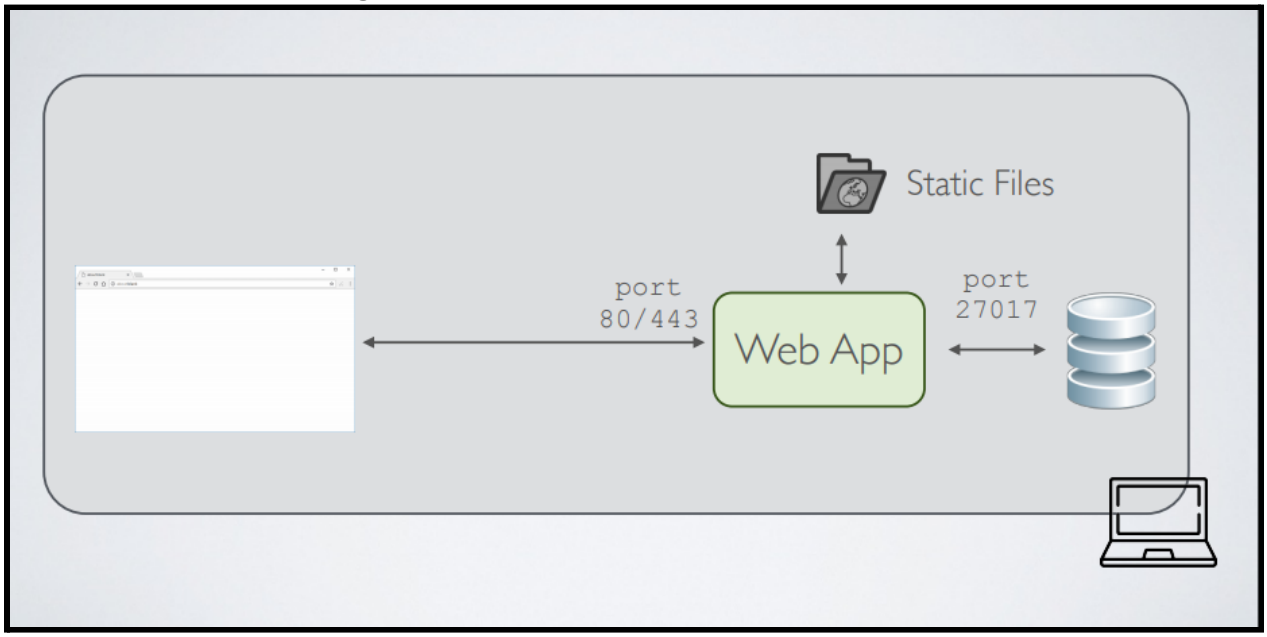
Then, the browser will then make the actual request. If service.example.com does not accept cross-site requests from this origin then it will respond with error to the OPTIONS request and the browser will not make the actual request.

2. **Simple Requests:**
   - A **simple request** is a CORS request that doesn't require a preflight request before being initiated.

- **JSON with Padding (JSONP)** is another way to circumvent same-origin policy. JSONP is a historical JavaScript technique for requesting data by loading a <script> element. Because JSONP is vulnerable to CSRF attacks, it is very dangerous to use and has been replaced with CORS.
- We can protect legitimate requests with a CSRF token.
- We can also prevent CSRF attacks using the SameSite cookie flag. If you're using the SameSite flag, the cookie will not be sent over cross-site requests. This prevents forwarding the authentication cookie over cross origin requests.

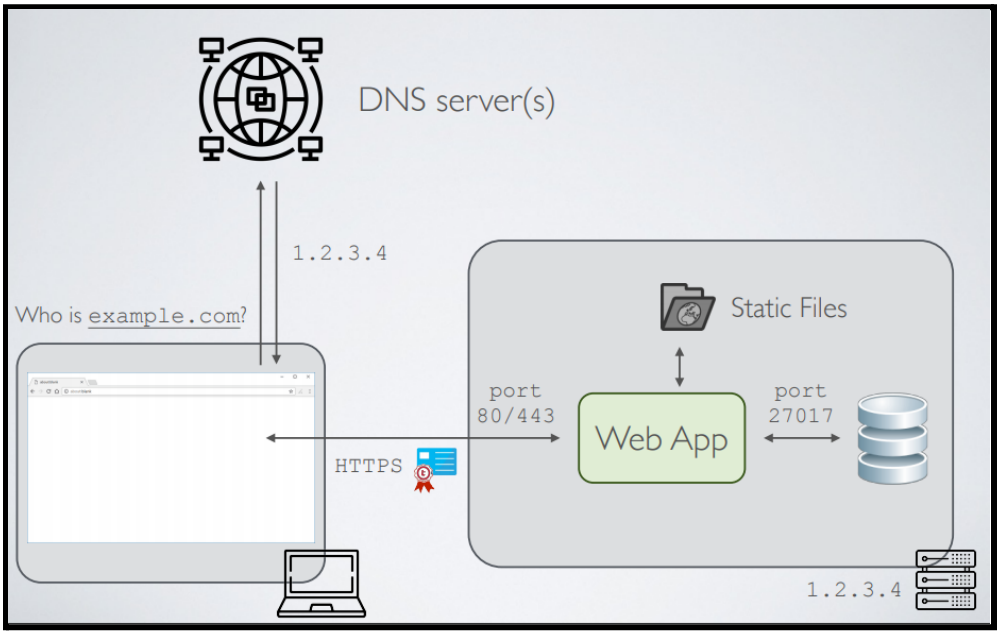
Deploying Web Applications:

- **DNS Server:**
- The **Domain Name System (DNS)** is the phonebook of the Internet. When users type domain names such as ‘google.com’ or ‘nytimes.com’ into web browsers, the DNS is responsible for finding the correct IP address for those sites. Browsers then use those addresses to communicate with origin servers or CDN edge servers to access website information. This all happens thanks to DNS servers: machines dedicated to answering DNS queries.
- **What We Have Been Doing:**



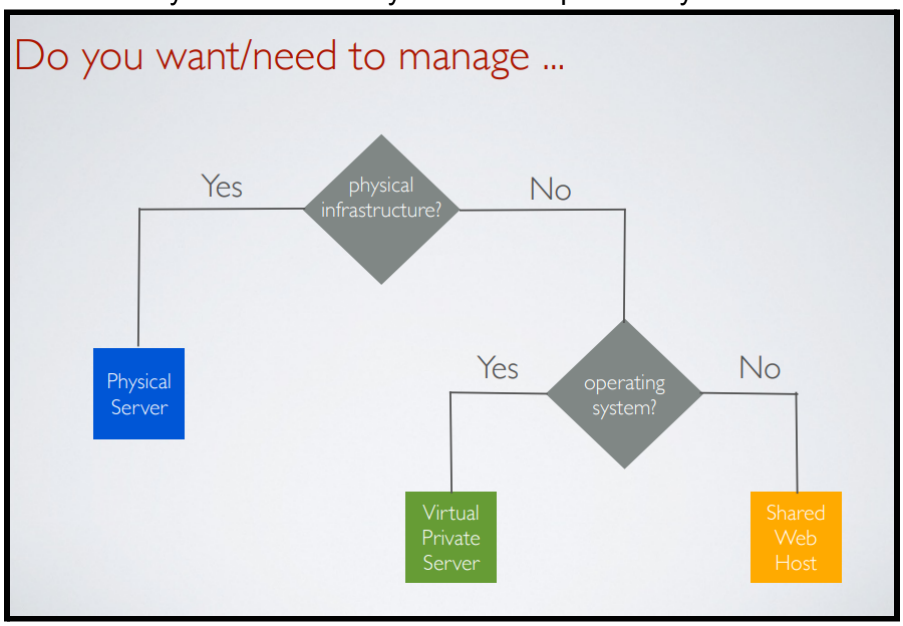
So far, we’ve been running both the frontend and backend on our laptops/PC.

- **What We Want To Do:**



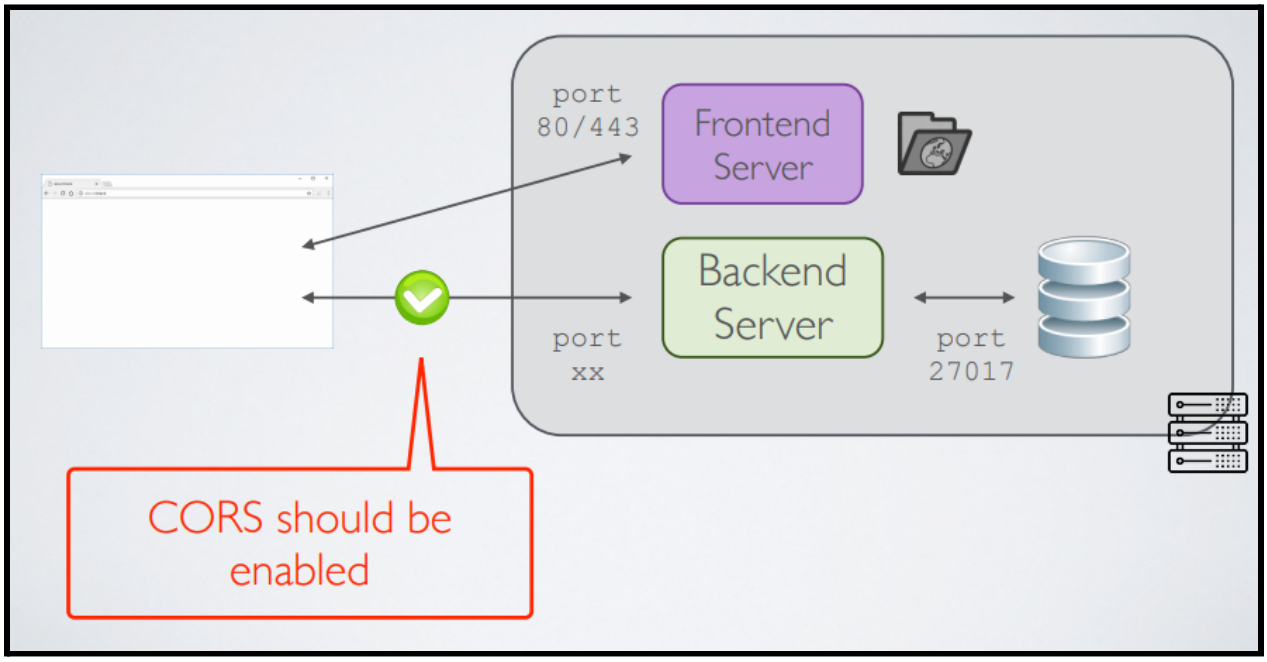
We want to deploy our web application (backend) on a web server that’s remote and has a fixed IP address. We also need a domain name and set up HTTPS on the server.

- **What We Need:**
    1. **Web Host:** A server to host your website.
    2. **Domain Name:** A url for your website.
    3. **Valid Certificate:** A signed certificate for HTTPS.
- **Web Hosting:**
- Most web frameworks provide a development server. These development servers may not be production ready and might not scale with multiple requests (multi-threading).
- **Note:** Node.js is production ready.
- Choosing a web host will depend on many things:
    1. Processing Power: How much CPU and RAM do you need?
    2. Storage: How much space do you need?
    3. Bandwidth: How much traffic do you expect?
    4. Money: How much do you want to spend daily



- Choosing a hosting solution depends on:
    1. Specific needs: Specific applications that your web applications use.
    2. Security: What you are comfortable to administer.
- Dedicated Physical Server:
    - You have total control.
    - However, you have to worry about the maintenance of the physical infrastructure, administration of the operating system.
    - It is not flexible. If you need more power, you need to buy more RAM, CPU, computers, etc.
- Virtual Private Server (VPS):
    - You have to worry about the administration of the operating system.
    - No maintenance of the physical infrastructure. Someone else will take care of this.
    - Flexibility (pay for what you need).
    - E.g. AWS
- Shared Web Host:
    - No administration of the operating system.
    - Very expensive.
    - Not adequate for specific needs.

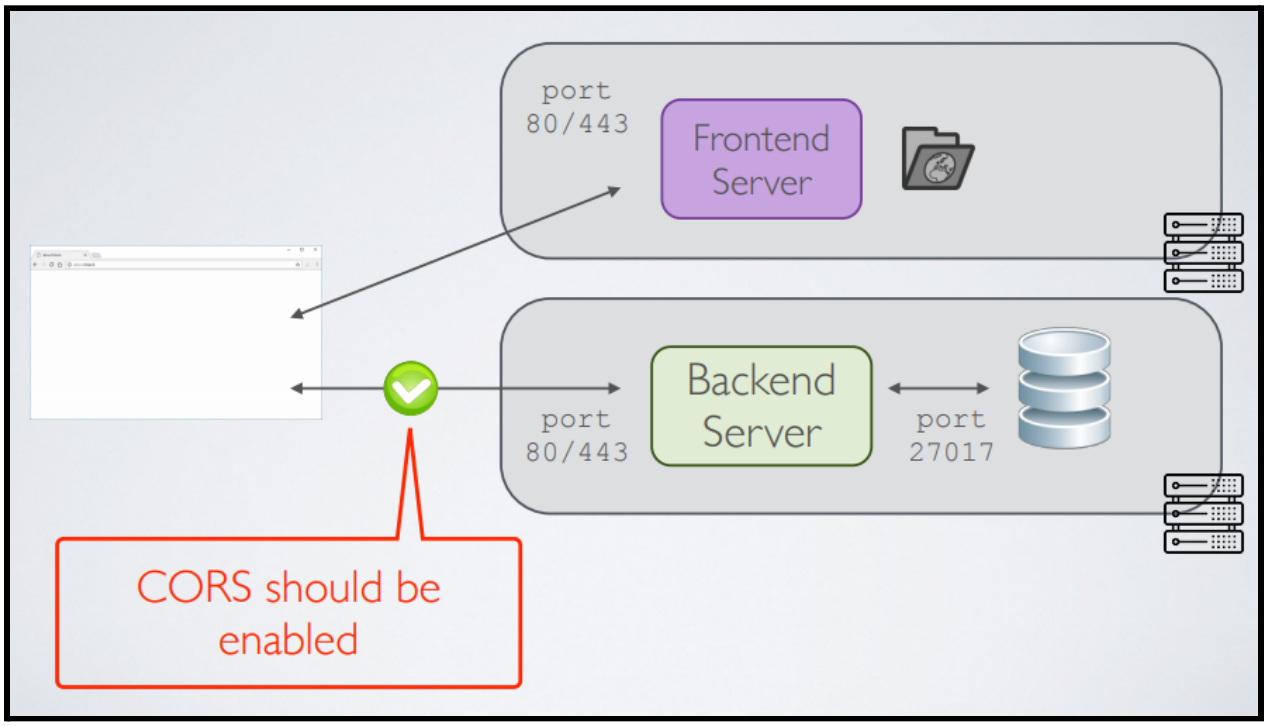
- **Deploying on physical or virtual server:**
- Two servers on the same host



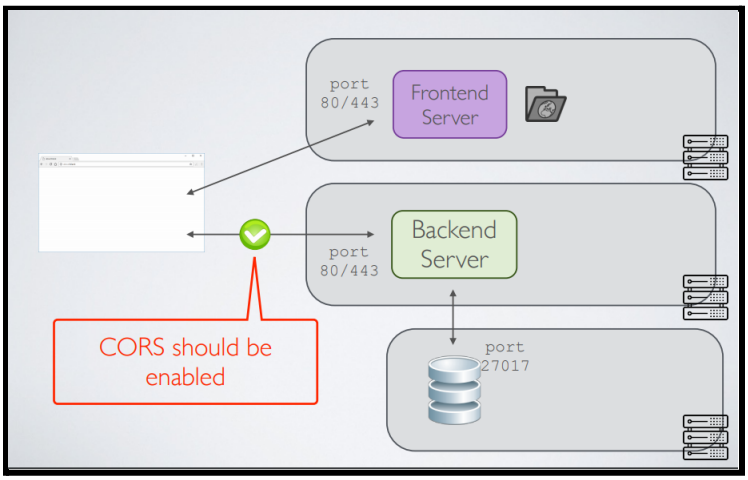
Note: 2 drawbacks of this method are:

1. The database port may be exposed.
2. The packages you installed for the frontend and backend may conflict and cause weird things to happen.

- Two servers on different hosts

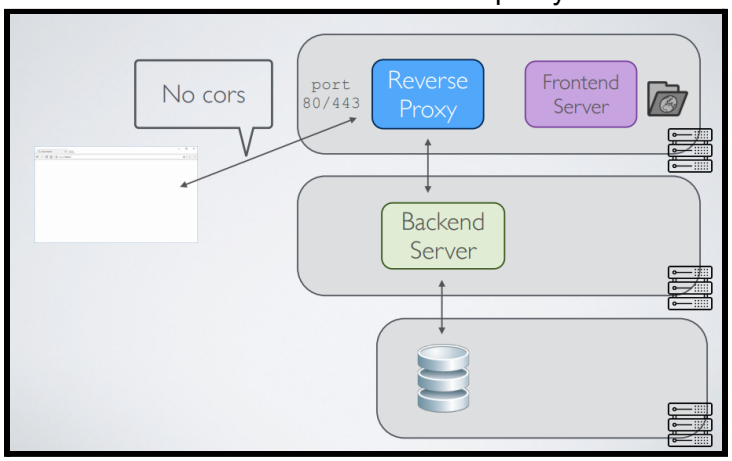


- Three-tiered architecture on different hosts

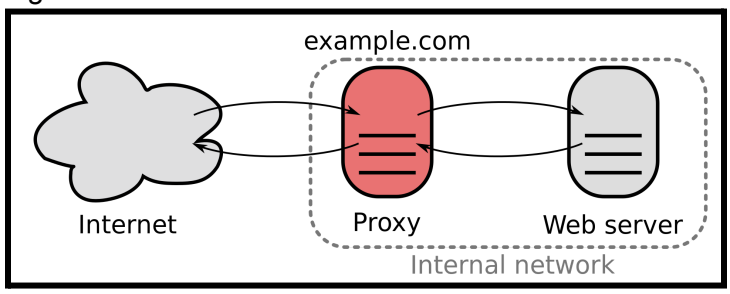


If your frontend crashes, your backend is still running and vice versa.
A drawback is now you need 3 servers.
Another drawback is now enabling CORS can be annoying.

- Three-tiered architecture with reverse proxy

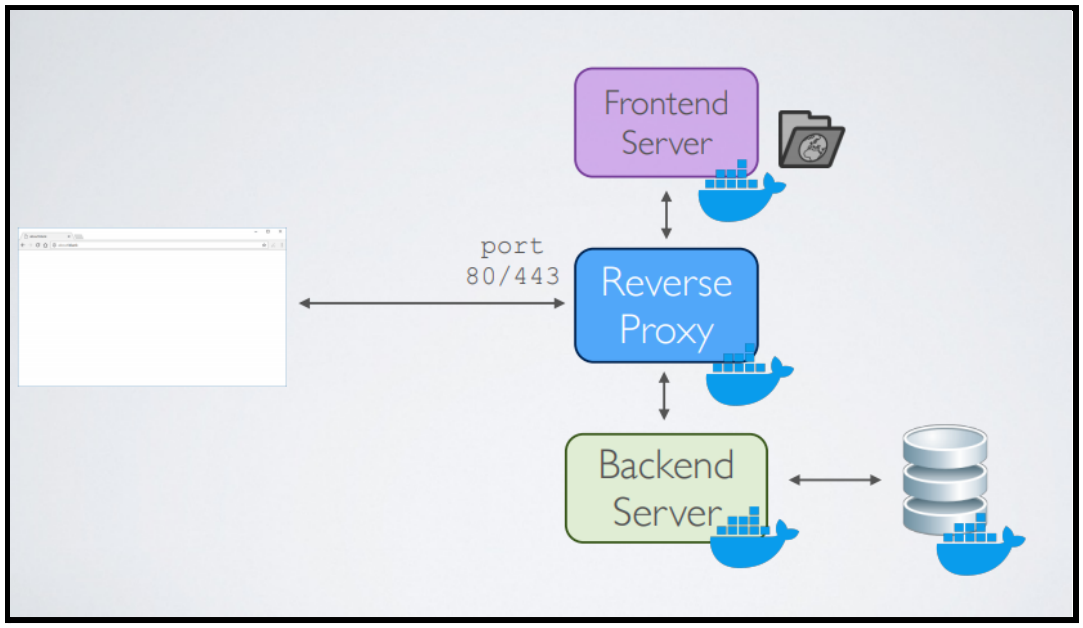


The reverse proxy acts as a gateway/router.
E.g.

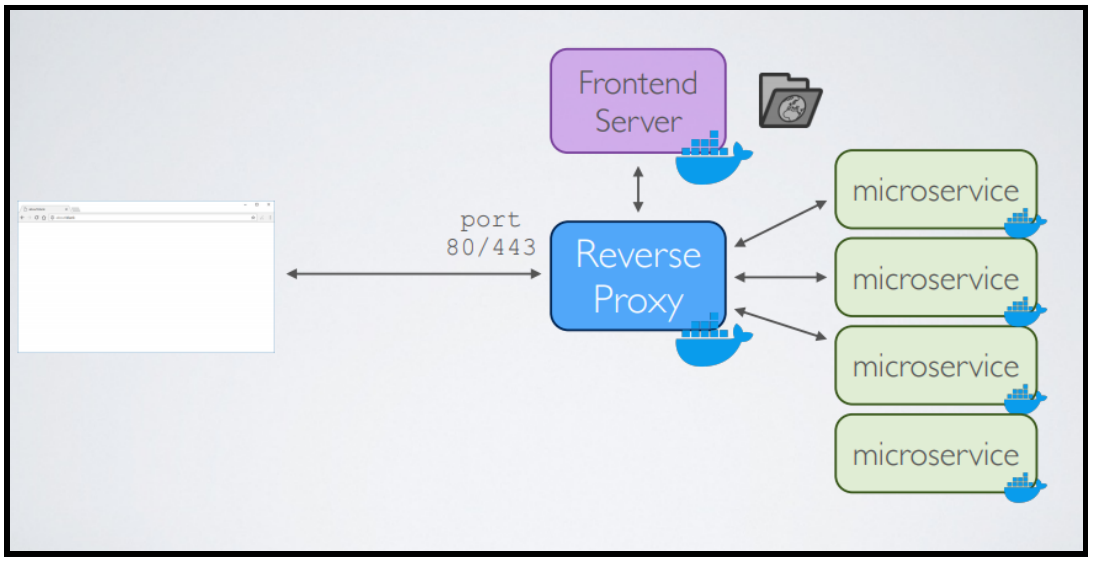


A client makes a request to a reverse proxy server. The proxy inspects the request, determines that it is valid and that it does not have the requested resource in its own cache. It then forwards the request to some internal web server. The internal server delivers the requested resource back to the proxy, which in turn delivers it to the client. The client is unaware of the internal network, and cannot tell whether it is communicating with a proxy or directly with a web server.

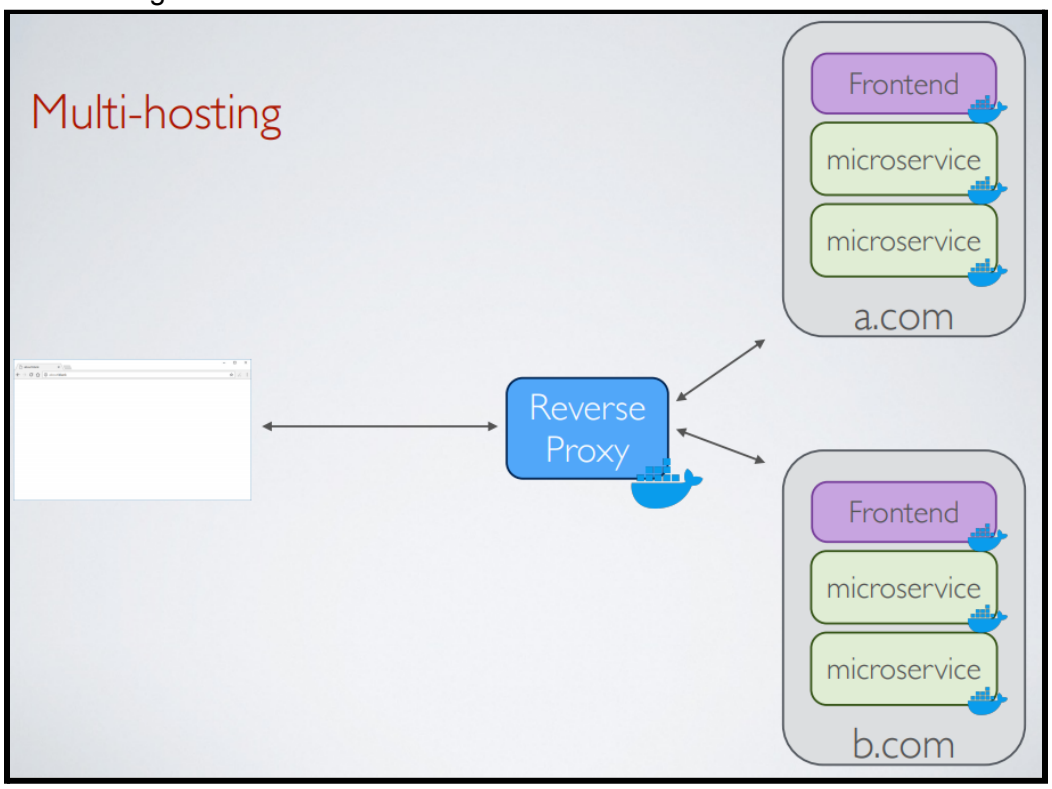
- Why have separated servers:
  - Each piece of our three-tiered architecture relies on specific OS configurations, libraries and runtime environment. These environments might conflict with each other. Having several servers isolates them, making sure they won't conflict with each other. It is easier to maintain and more reliable. Furthermore, if one server (frontend/backend/database) crashes, it won't affect the other parts.
  - However having several servers has a cost. A solution to this is to use virtual servers or containerized servers. It is cost effective and even simpler to maintain and to scale. You can have virtual machines (VMs) inside other VMs.
    One popular containerized server is Docker.
- Dockerized three-tiered architecture



- Dockerized micro-service architecture



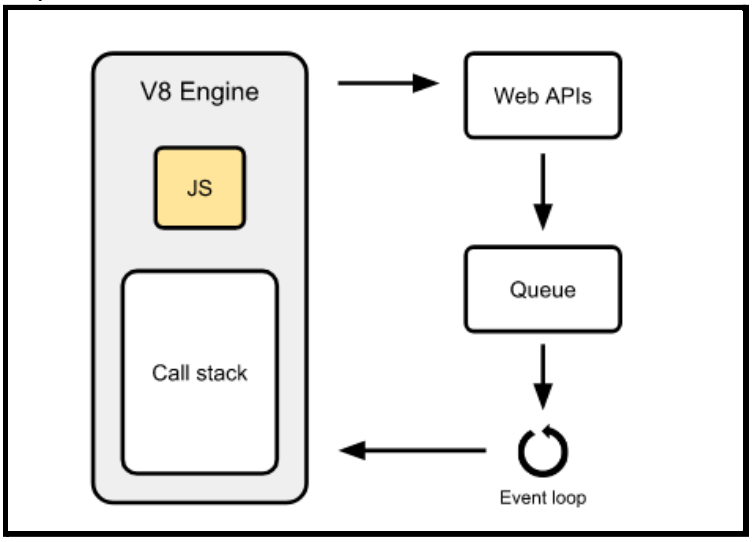
- Multi-hosting:



- **Domain Name:**
- Top Level Name:
    - A **top level name** is the last part of the URL. It is the .com or .ca or .edu, etc. Here is a list of top level names: https://en.wikipedia.org/wiki/List_of_Internet_top-level_domains. Note that not all top level names are available for purchase. .edu is reserved for educational facilities like a university and .gov is reserved for governments.
    - Each country also has a top level name. For Canada, it is .ca.
- To get a domain name, you need to buy one from a **domain name registrar**. Domain name registrars are companies that sell domain names.
  Examples of domain name registrar are GoDaddy and Namecheap.
  Note: You have to renew the domain name. Otherwise, other people can buy it.
- **WHOIS** is a query and response protocol that is widely used for querying databases that store the registered users or assignees of an Internet resource, such as a domain name. I.e. WHOIS gets information about a website.
- **A Valid Certificate:**
- The domain name registrar can provide a valid certificate.
- There are also specific companies that provide certificates. You need to be able to prove to them that you own the website.

Javascript Execution Model (The Event Loop):

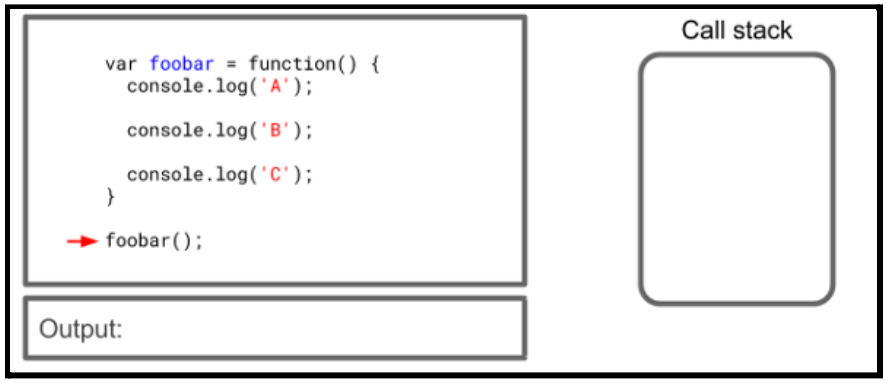
- **Blocking code** is code that runs one instruction after another, and makes next instructions wait.
- **Non-blocking code** means that we don't have to wait for some code to run before running other code.
  I.e. Blocking refers to operations that block further execution until that operation finishes while non-blocking refers to code that doesn't block execution.
- Blocking methods execute synchronously and non-blocking methods execute asynchronously.
- There are two types of function calls:
  - Synchronous calls are pushed to the call stack.
  - Asynchronous calls are pushed to the event queue.
- Example of asynchronous calls are:
  - DOM events (browser)
  - Ajax requests (browser)
  - Timer (browser and NodeJs)
  - Any non-blocking I/O (NodeJs)
  - Promises and async/await
- JavaScript must be compiled and interpreted. It needs a runtime environment.
  In Chrome, the JS runtime is the V8 Engine. Furthermore, Javascript is an event-driven language. It uses the **event loop** to keep track of all of these events. The event loop is a way of scheduling events one after the other. It often gets help from the platform the engine is running on. It is important to note that JS is single-threaded, but that browsers are multi-threaded. Non-blocking JS functions aren't built into the V8 engine. They live in the platform JS is running on, the browser. For example, Chrome contains the instructions for setTimeout, a non-blocking function in JS.
- V8 is a popular open source JS Engine. It is the engine inside Chrome that runs the JavaScript code. But when running JavaScript in the browser, there's more involved than just the JS Engine. Most JavaScript code makes use of web APIs like DOM, AJAX, and Timeout (setTimeout). These APIs are provided by the browser.
- In the picture below we can see an overview of the elements involved in the JS execution in the browser. The V8 Engine contains a call stack. The engine communicates with the web APIs, and there is also a queue and a mysterious event loop.



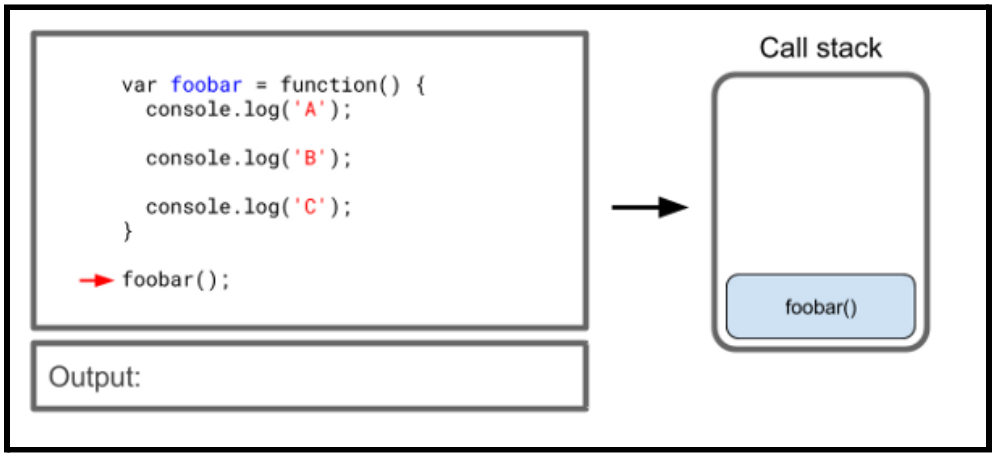
- Javascript is a single threaded programming language, which means it has a single call stack and can do one thing at a time.
- The call stack is a mechanism so the interpreter knows its place in a script. When a script calls a function, the interpreter adds it on the top of the call stack. When the current function is finished, the interpreter takes it off the stack.
- Some terminology:
    - **Call stack:** It's a simple data structure that records where in the code we are currently. So if we step into a function that is a function invocation it is pushed to the call stack. When we return from a function it is popped out of the stack.
    - **Heap:** It's where objects are allocated. Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.
    - **Queue/Callback Queue:** It stores all the callbacks.
      It stores a list of messages to be processed. Each message has an associated function which gets called in order to handle the message.
      At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. To do so, the message is removed from the queue and its corresponding function is called with the message as an input parameter. As always, calling a function creates a new stack frame for that function's use.
      The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue if there is one.
    - **Event Loop:** This is where all these things come together. The event loop simply checks the call stack, and if it is empty (which means there are no functions in the stack) it takes the oldest callback from the callback queue and pushes it into the call stack which eventually executes the callback.
- E.g. Consider this example:
  We have this code:

```
1  var foobar = function() {
2    console.log('A');
3    console.log('B');
4    console.log('C');
5  }
6  foobar();
```

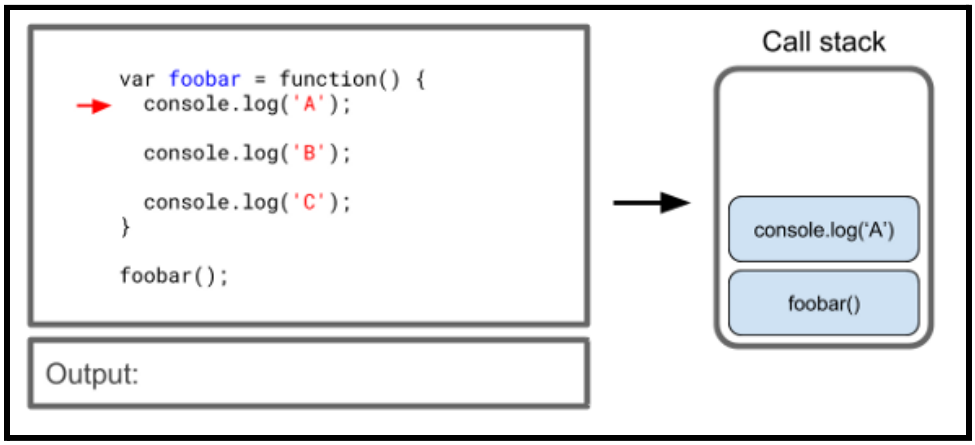
Step 1:
The interpreter starts calling foobar.



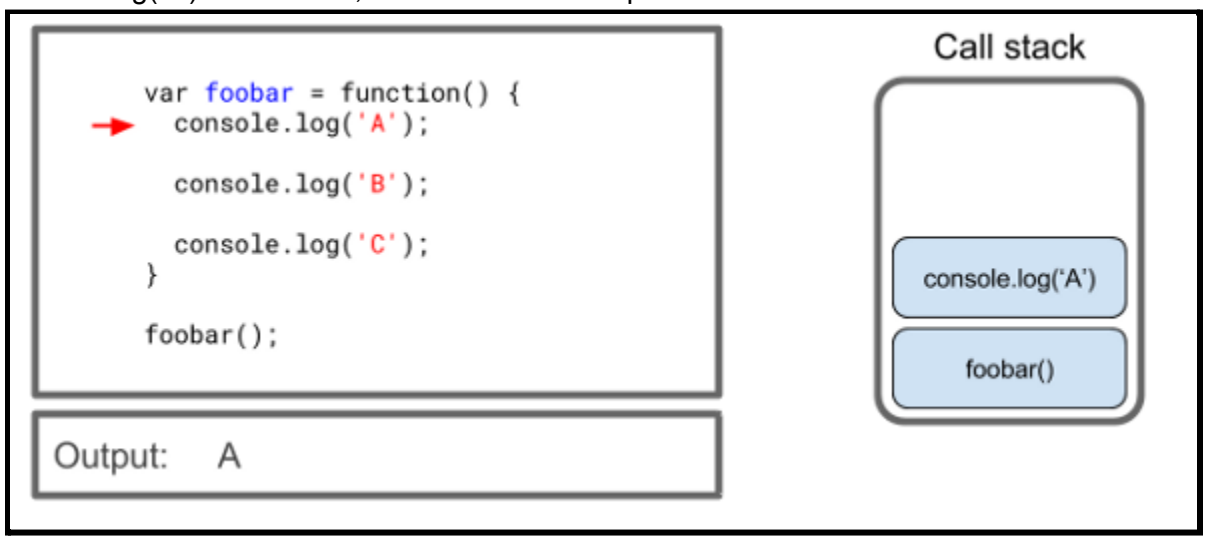
Step 2:
foobar is added to the stack.



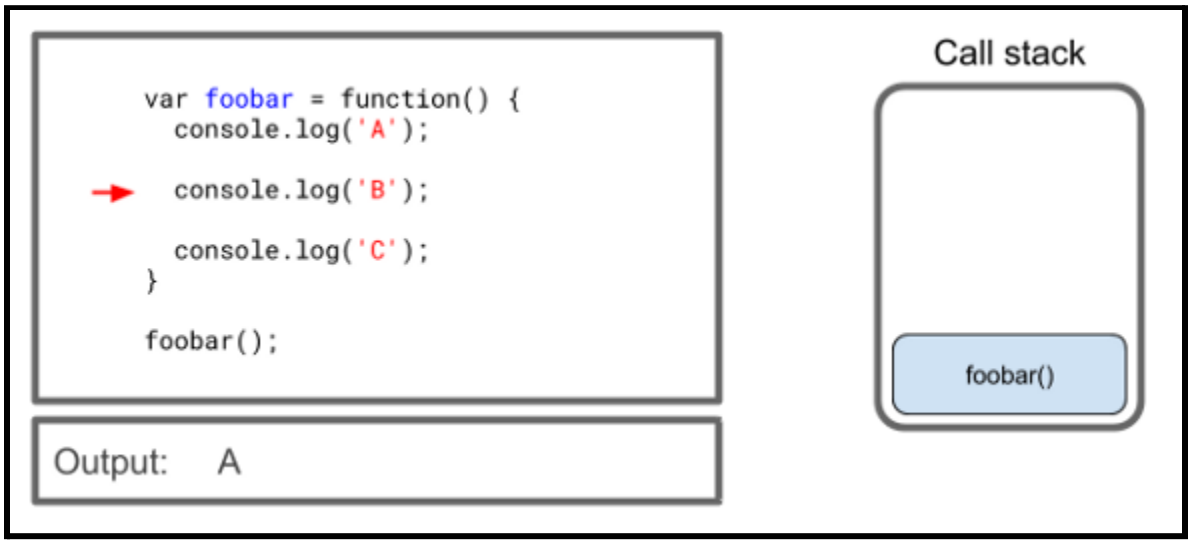
Step 3:
The interpreter steps into the function, and console.log('A') is called and added to the top of the call stack.



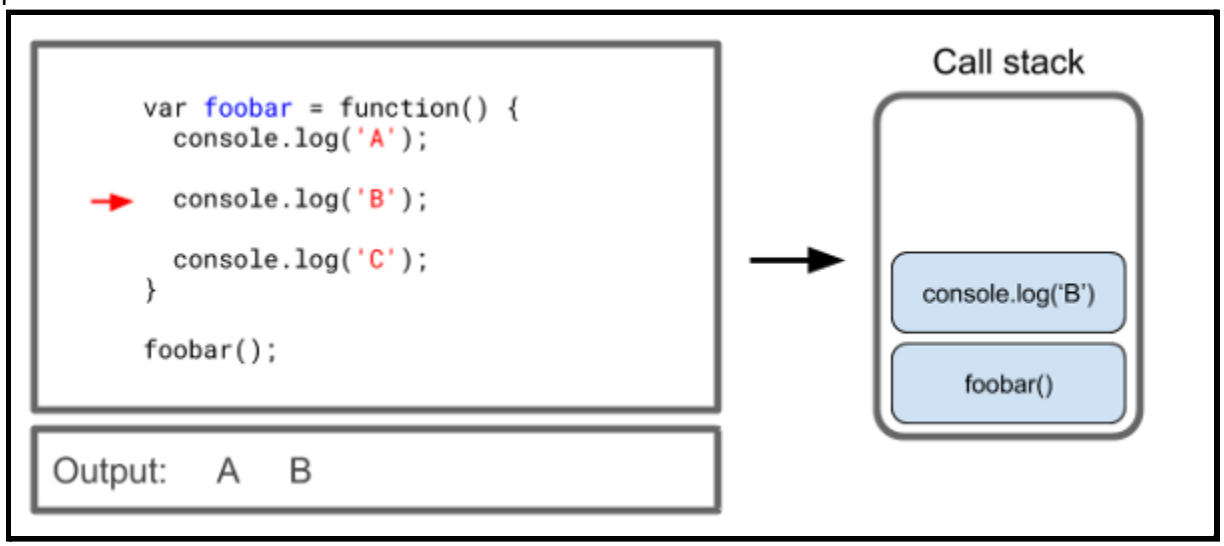
Step 4:
console.log('A') is executed, and the letter "A" is printed in the console.



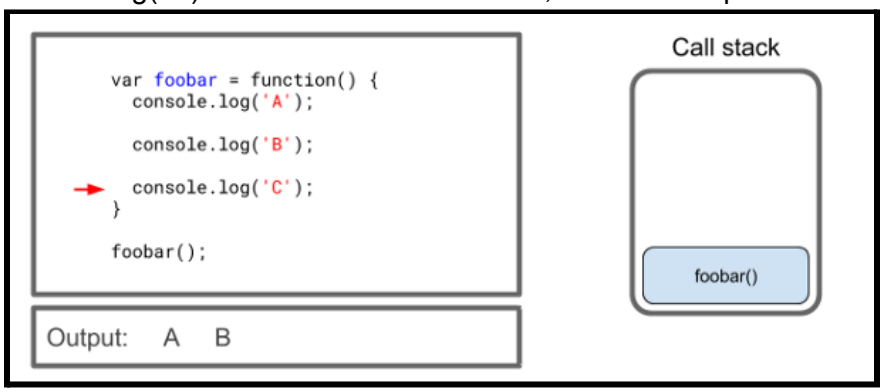
Step 5:
console.log('A') is removed from the call stack since its execution has been completed, and the interpreter moves on to the next command.



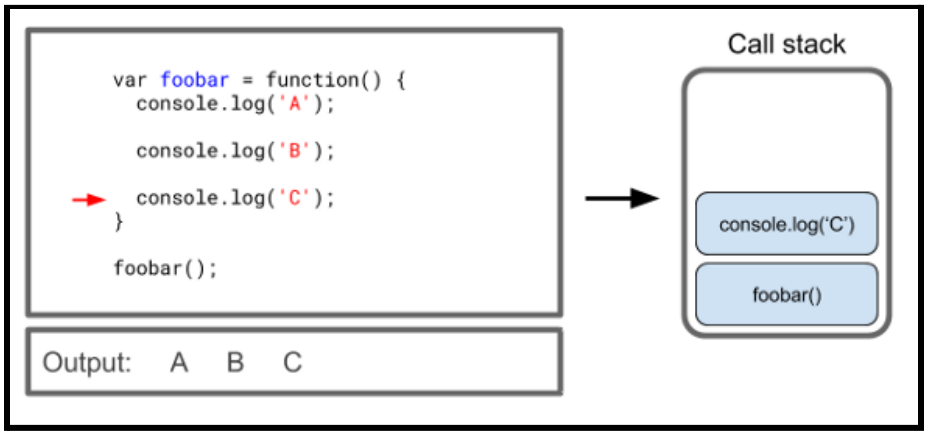
Step 6:
console.log('B') is processed the same way as the previous one, and the letter 'B' is printed in the console.



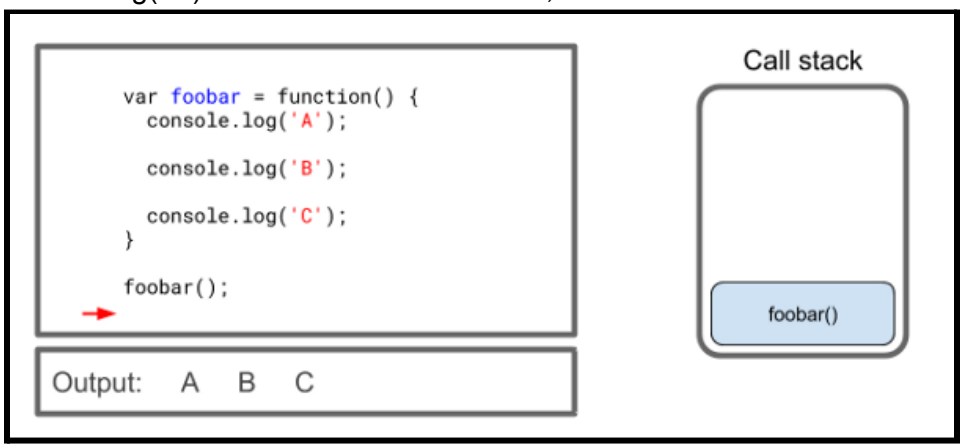
Step 7:
console.log('B') is removed from the stack, and the interpreter moves on.



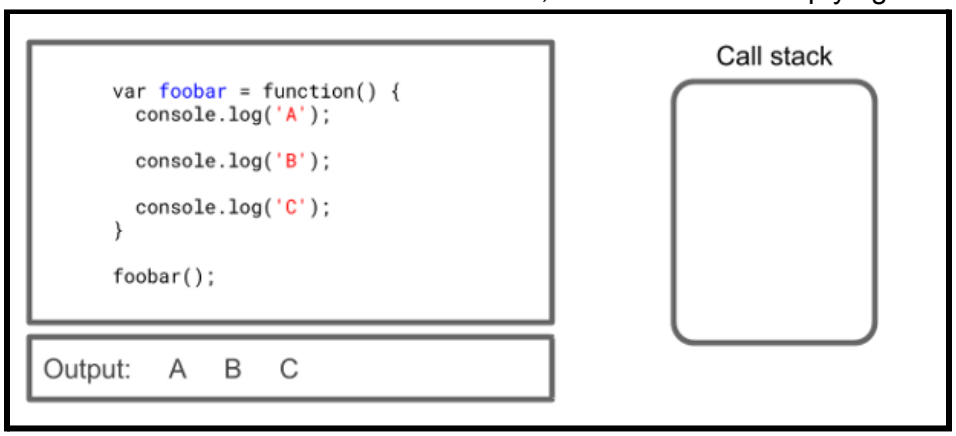
Step 8:
console.log('C') is processed the same way as the previous ones, and the letter 'C' is printed in the console.



Step 9:
console.log('C') is removed from the stack, and the end of the foobar function is reached.



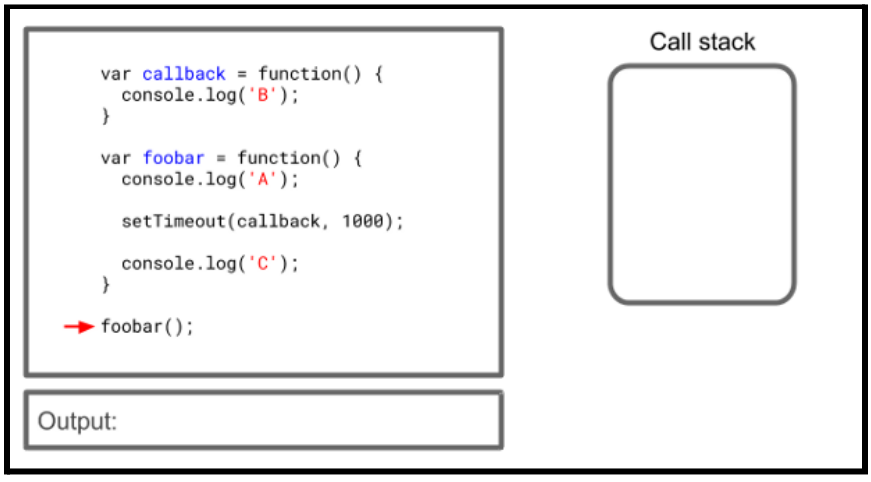
Step 10:
foobar is also removed from the call stack, which becomes empty again.



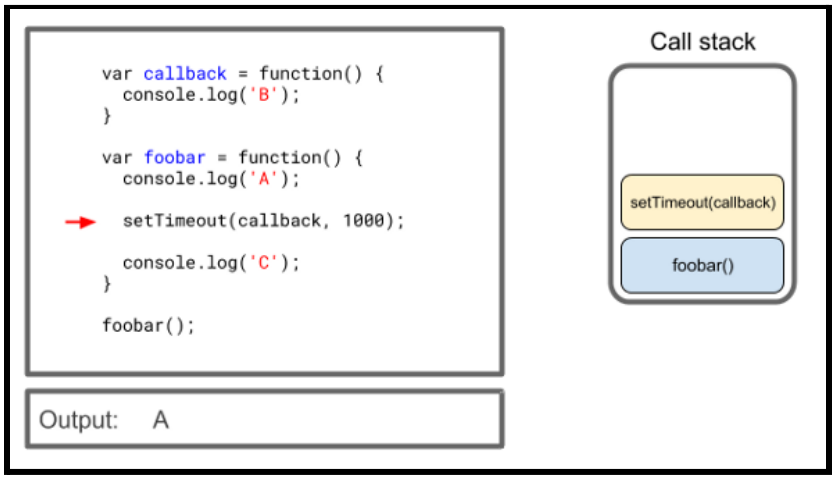
- E.g. Consider this example:
We have this code:

```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

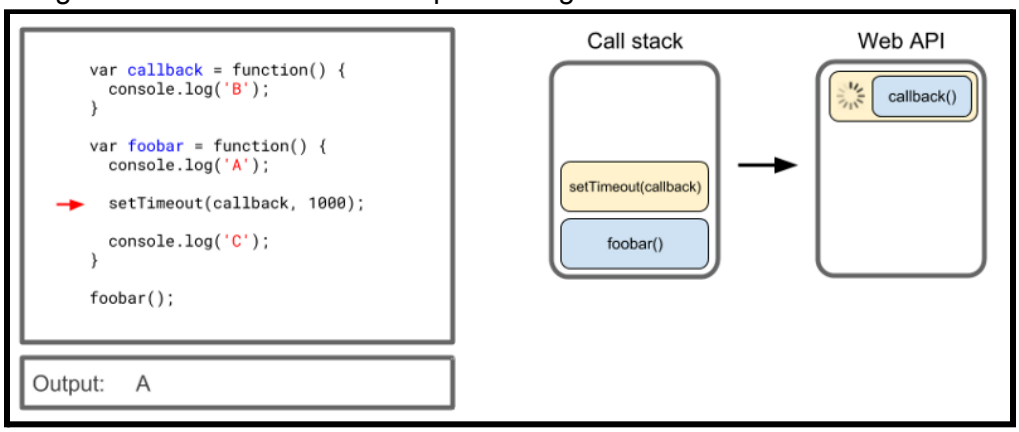
Note: This time, instead of directly doing “console.log(“B”);” in foobar, we have a callback function that does it after waiting 10 seconds.
Step 1:
Now, instead of calling console.log(‘B’) directly, we are putting it inside a callback function that will be called later. The setTimeout is an asynchronous function that we are using to simulate a slow operation. Here again, the interpreter starts by calling foobar.



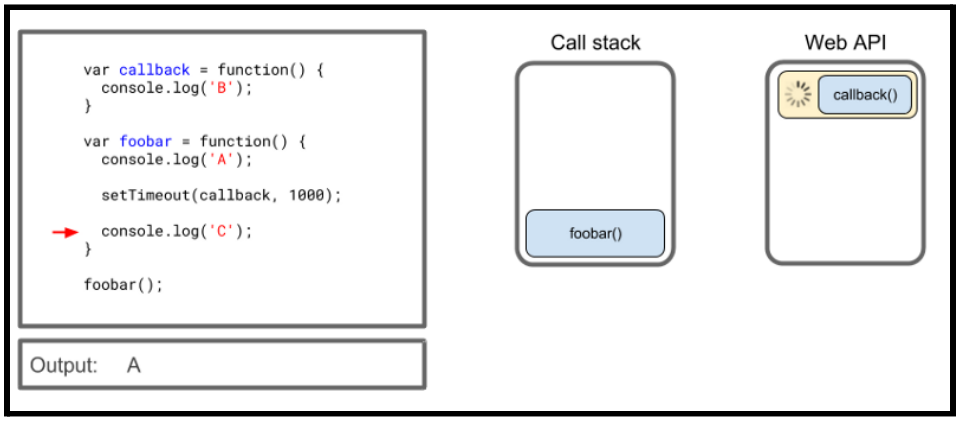
Step 2:
The first steps are the same: foobar is added to the call stack, the interpreter steps in, adds console.log('A') to the stack, prints the letter 'A,' and removes the call from the stack. Then the interpreter reaches the setTimeout function and adds it to the call stack.



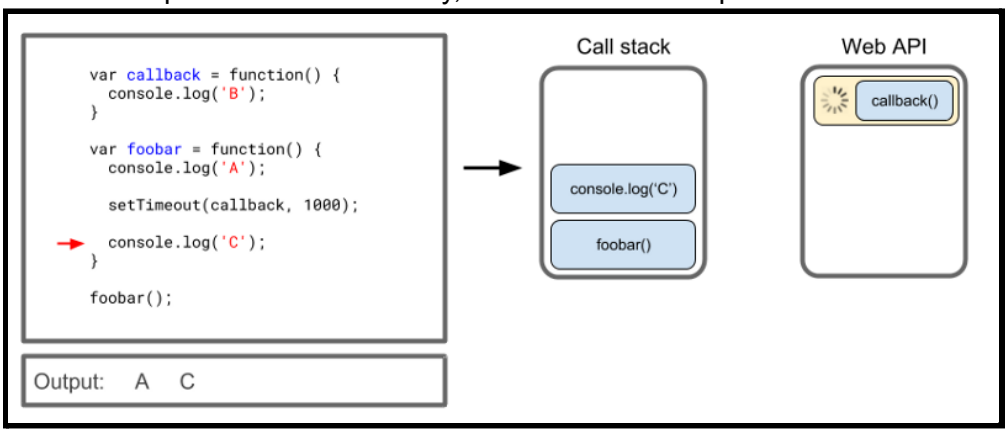
Step 3:
Since setTimeout is provided for us by the Web API, it receives the setTimeout call alongside the callback and starts processing it.



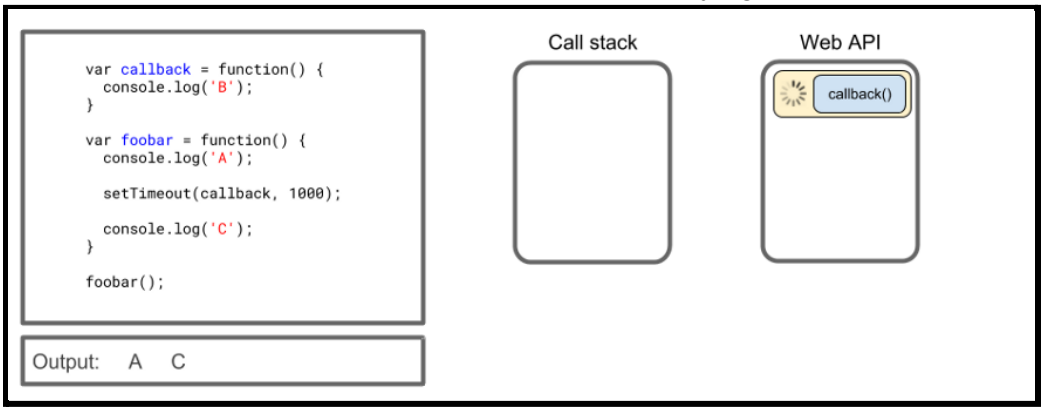
Step 4:
Now the call to setTimeout itself is completed, so it can be removed from the stack. The Web API continues the processing, but the call stack is not blocked anymore, so the interpreter can move to the next command.



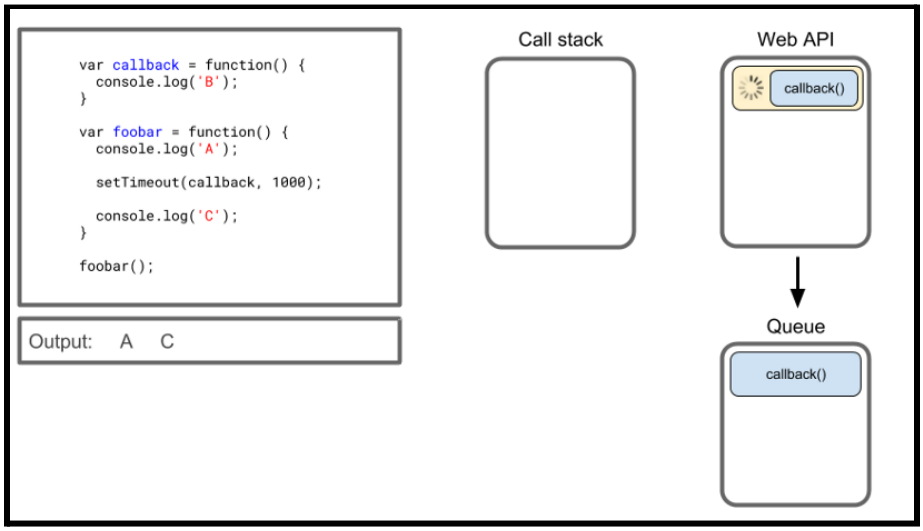
Step 5:
This next step is executed normally, and the letter 'C' is printed to the console.



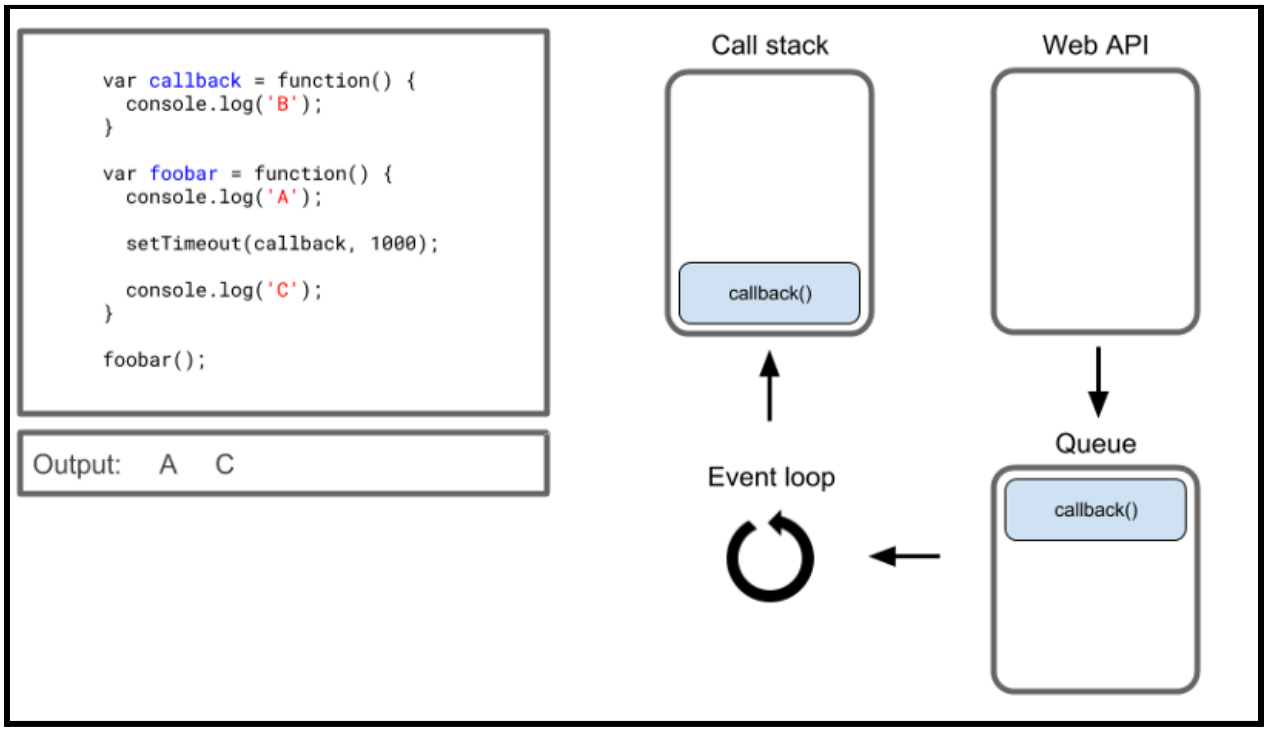
Step 6:
console.log('C') is removed from the call stack, and so is foobar since the interpreter has reached the end of the function. The call stack is empty again.



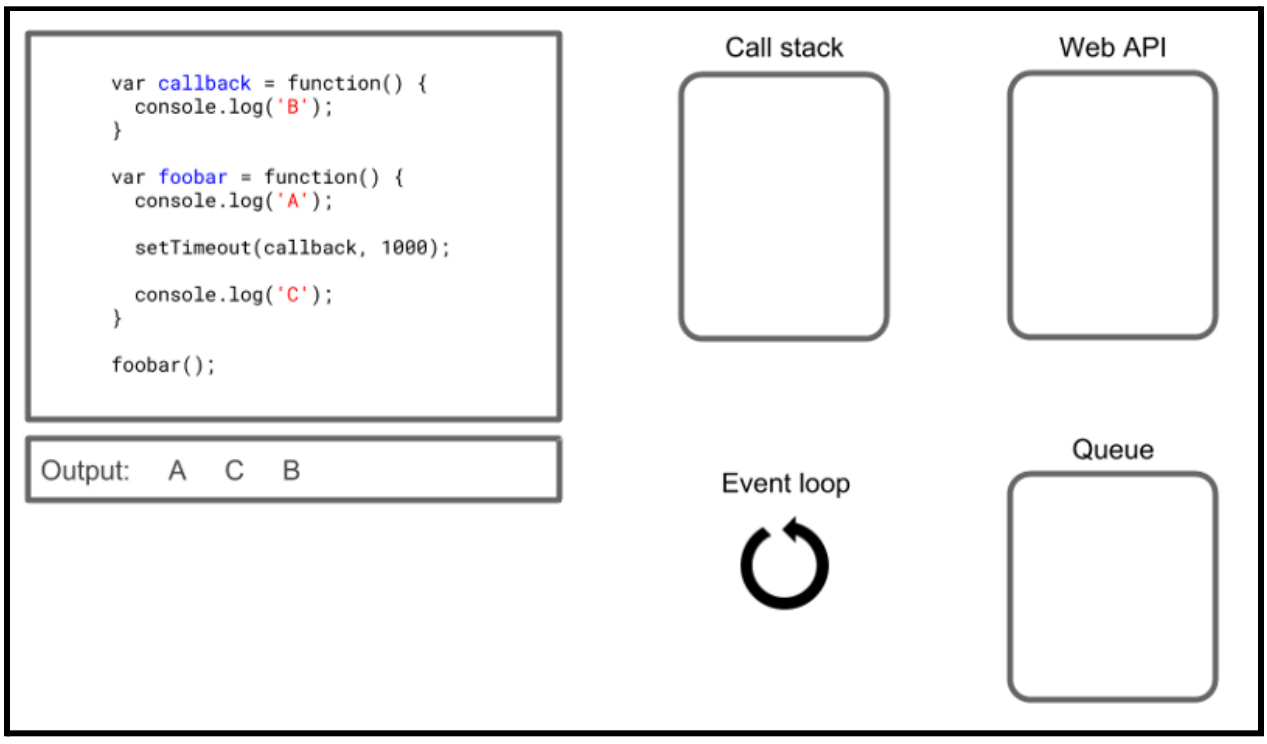
Step 7:
When an asynchronous call is completed, the callback function is pushed to the queue.



Step 8:
This is where the last piece of the puzzle comes in--the event loop. The event loop has one simple job: it looks at the call stack and the task queue, and if the stack is empty, it takes the first item in the queue and sends it back to the call stack.



Step 9:
Finally, the callback is executed, the letter "B" is printed in the console, and the callback is removed from the call stack.



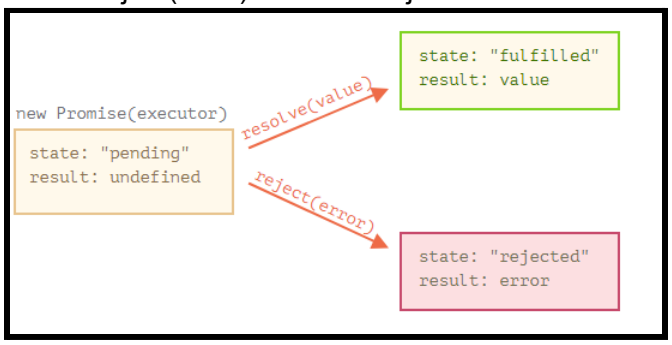
- Here is a good resource that lets you visualize the process: http://latentflip.com/loupe/.
- **Note:** The event loop will look at the queue if the stack is empty.

Multi-Threaded vs Single-Threaded:

- Multi-threading does not necessarily mean things are executed in parallel. Remember, we only have one CPU.
- We need multi-threading because programming languages have blocking I/O, and by default, programs wait for the I/O to be completed.
- But multi-threading is expensive in terms of software design (synchronization) and in terms of performances (context switch).
- An alternative to multi-threading is single-threaded with non-blocking I/O.
- Running a single-threaded server will have good performance, as long as the requests handlers:
    - Do some asynchronous I/O such as filesys, database, cache, network, etc.
    - Do not do any heavy but yet synchronous computations such as complex math, intensive data processing, etc.
- If needed, Javascript can be multi-threaded.
- Examples of multi-threaded Javascript are Node cluster (NodeJS only) and Web Workers (Browser and NodeJS). They are good for heavy but yet synchronous computations and take advantage of multicore machines.

Dealing with Asynchronism:

- **Promises:**
- The **Promise** object represents the eventual completion or failure of an asynchronous operation and its resulting value.
- A Promise is in one of these states:
    1. pending: initial state, neither fulfilled nor rejected.
    2. fulfilled: meaning that the operation was completed successfully.
    3. rejected: meaning that the operation failed.
- A pending promise can either be fulfilled with a value or rejected with a reason (error).
- Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.
- As the Promise.prototype.then() and Promise.prototype.catch() methods return promises, they can be chained.
- A pending promise may transition into a fulfilled or rejected state.
- A fulfilled or rejected promise is settled, and must not transition into any other state.
- Once a promise is settled, it must have a value (which may be undefined). That value must not change.
- The arguments resolve and reject are callbacks provided by JavaScript itself. Our code is only inside the executor. When the executor obtains the result, it should call one of these callbacks:
    1. resolve(value) - State if fulfilled.
    2. reject(error) - State is rejected.



- **Note:** Even with resolve or reject, the value could still be undefined.
- E.g.

```
const a = 3;
const b = 3;

let p = new Promise((resolve, reject) => {
    if (a === b){
        resolve("true");
    }
    else{
        reject("false");
    }
})

console.log(p);

p.then((message) => { // Runs if resolve
    console.log("a === b is " + message);
}).catch((error) => { // Runs if reject
    console.log("a === b is " + error);
})
```

```
Promise { 'true' }
a === b is true
```

Note: resolve and reject are callback functions.
Here, since a === b, the promise runs resolve function, which will run the .then function. Furthermore, we see that the promise is fulfilled.

- E.g.

```
const a = 3;
const b = 4;

let p = new Promise((resolve, reject) => {
    if (a === b){
        resolve("true");
    }
    else{
        reject("false");
    }
})

console.log(p);

p.then((message) => { // Runs if resolve
    console.log("a === b is " + message);
}).catch((error) => { // Runs if reject
    console.log("a === b is " + error);
})
```

```
Promise { <rejected> 'false' }
a === b is false
```

Here, since a !== b, the promise runs resolve function, which will run the .catch function. Furthermore, we see that the promise is rejected.

- E.g.

```
const a = 3;
const b = 3;

let p = new Promise((resolve, reject) => {
    return("true");
})

console.log(p);

p.then((message) => { // Runs if resolve
    console.log("a === b is " + message);
}).catch((error) => { // Runs if reject
    console.log("a === b is " + error);
})
```

```
Promise { <pending> }
```

Here, since we don't use resolve or reject, the promise is pending.

- E.g.

```
const a = 3;
const b = 3;

let p = new Promise((resolve, reject) => {
    if (a === b){
        resolve();
    }
    else{
        reject();
    }
})

console.log(p);

p.then((message) => { // Runs if resolve
    console.log("a === b is " + message);
}).catch((error) => { // Runs if reject
    console.log("a === b is " + error);
})
```

```
Promise { undefined }
a === b is undefined
```

Here, because resolve doesn't give any value, we get undefined.

- **Async/Await:**
- The keyword **async** before a function makes the function return a promise.
- E.g.

```
async function foo() {
  return 1;
}
```

is equivalent to

```
function foo() {
  return Promise.resolve(1);
}
```

- E.g.

```
let p1 = function(){
    return 1;
}

let p2 = async function(){
    return 1;
}

console.log(p1, p1());
console.log(p2, p2());
```

```
[Function: p1] 1
[AsyncFunction: p2] Promise { 1 }
```

- The keyword **await** before a function makes the function wait for a promise.
- The await keyword can only be used inside an async function.
- The await expression causes an async function execution to pause until a Promise is settled (that is, fulfilled or rejected), and to resume execution of the async function after fulfillment. When resumed, the value of the await expression is that of the fulfilled Promise.
- If the Promise is rejected, the await expression throws the rejected value.
- If the value of the expression following the await operator is not a Promise, it's converted to a resolved Promise.
- An await splits the execution flow, allowing the caller of the async function to resume execution. After the await defers the continuation of the async function, execution of subsequent statements ensues. If this await is the last expression executed by its function, execution continues by returning to the function's caller a pending Promise for completion of the await's function and resuming execution of that caller.
- await can be put in front of any async promise-based function to pause your code on that line until the promise fulfills/rejects, then return the resulting value.
- You can use await when calling any function that returns a Promise, including web API functions.
- The body of an async function can be thought of as being split by zero or more await expressions. Top-level code, up to and including the first await expression (if there is one), is run synchronously. In this way, an async function without an await expression will run synchronously. If there is an await expression inside the function body, however, the async function will always complete asynchronously.
- Code after each await expression can be thought of as existing in a .then callback. In this way a promise chain is progressively constructed with each reentrant step through the function. The return value forms the final link in the chain.
- E.g.

```
async function foo() {
  await 1;
}
```

is equivalent to

```
function foo() {
  return Promise.resolve(1).then(() => undefined);
}
```

- E.g.

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  let x = resolveAfter2Seconds(10);
  console.log(x);
  console.log("1");
}

f1();
console.log("2");
```

```
Promise { <pending> }
1
2
```

Notice that x is still pending (because of the setTimeout function) and that 1 is printed before 2.

- E.g.

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  let x = await resolveAfter2Seconds(10);
  console.log(x);
  console.log("1");
}

f1();
console.log("2");
```

```
2
10
1
```

Now, when we use await, we see that we wait for resolveAfter2Seconds to return a settled value, either fulfilled or rejected, before continuing. Furthermore, notice that 2 is printed before 10 and 1. This is because await causes the code to run asynchronously. So while it's waiting for the result of resolveAfter2Seconds, the other parts of the code, not in f1, will run. Hence, 2 is printed first.

Web Workers:

- **Web workers** makes it possible to run a script operation in a background thread separate from the main execution thread of a web application. The advantage of this is that laborious processing can be performed in a separate thread, allowing the main (usually the UI) thread to run without being blocked/slowed down.
- Web workers create threads in Javascript (frontend and backend).
- These threads can run in parallel (separate event loop).

- What a web worker can/cannot do on the frontend

Can do	Cannot do
XMLHttpRequest	window
indexedDB	document (not thread safe)
location (read only)	

- Create a web worker

```
function(){
 "use strict";

 // receive message
 self.addEventListener('message', function(e){
   var data = e.data;
   // send the same data back
   self.postMessage(data);
 }, false);

});
```

- Instantiate a web worker

```
var worker = new Worker('doSomething.js');

// sending a message to the web worker
worker.postMessage({myList:[1, 2, 3, 4]});


// receive message from web worker
worker.addEventListener('message', function(e) {
     console.log(e.data);
}, false);
```

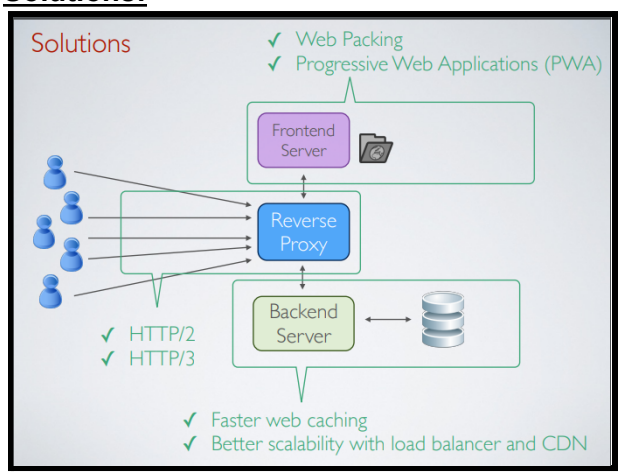
Users respond to speed:

- Amazon found every 100ms of latency cost them 1% in sales.
- Google found an extra .5 seconds in search page generation time dropped traffic by 20%.

Problems to consider:

- How to increase the throughput?
  Throughput refers to how much data can be transferred from one location to another in a given amount of time. It is used to measure the performance of hard drives and RAM, as well as Internet and network connections.
- How to scale to serve millions of users?

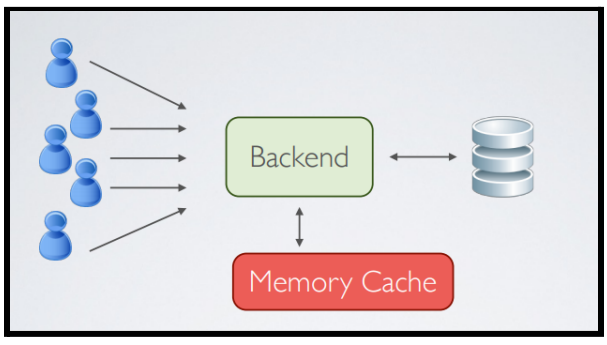
Solutions:



Backend Web Caching:

- Processing the request means:
  1. Parsing the HTTP request.
  2. Mapping the URL to the handler.
  3. Querying the database or third-party API.
  4. Computing the HTTP response.

  Of all these tasks, step 3 is the most expensive. DB and API accesses are expensive both in terms of time and money.
- **Caching:**
- In computing, a **cache** is a high-speed data storage layer which stores a subset of data, typically transient in nature, so that future requests for that data are served up faster than is possible by accessing the data's primary storage location. Caching allows you to efficiently reuse previously retrieved or computed data.
- The data in a cache is generally stored in fast access hardware such as RAM and may also be used in correlation with a software component. A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer.
- Instead of making the same queries to the database multiple times, we can store the information in cache to make it faster to retrieve in the future.
- I.e.



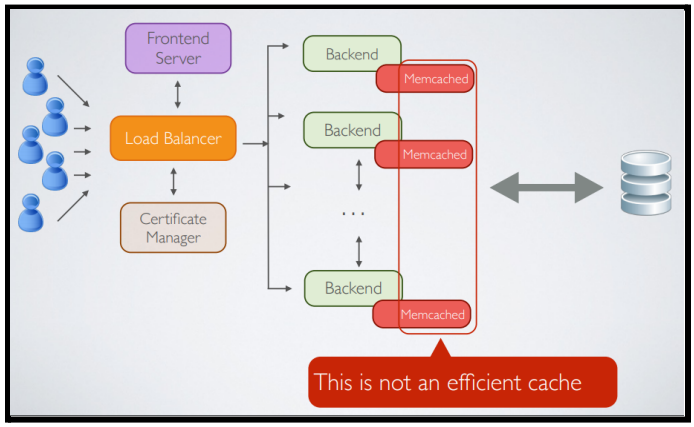
- The cache is controlled by the program and is specific for each app.
- We can cache database requests and session information.
- A popular memory cache is memcached.
- Memcached is a distributed shared cache.
- Memcached stores key/value pairs in memory and throws away data that is the least recently used.
- A typical cache algorithm:

```
retrieve from cache
if data not in cache:
        # cache miss
        query the database or API
        update the cache
return result
```

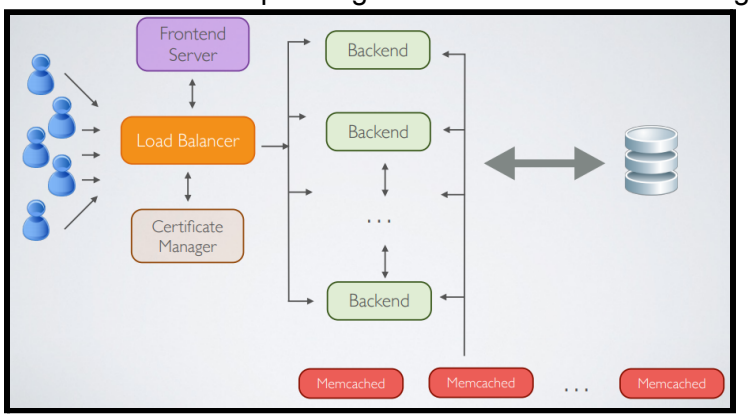
- **Cache stampede/dog piling** is a problem that occurs when multiple concurrent requests are doing the same request because cache was cleared. A cache stampede occurs when several threads attempt to access a cache in parallel. If the cached value doesn't exist, the threads will then attempt to fetch the data from the database at the same time. Due to the sudden spike in CPU usage, the database will crash.
- E.g.
  Imagine you are doing an expensive SQL query that takes 3 seconds to complete and spikes CPU usage of your database server. You want to cache that query as running multiple of those in parallel can cause database performance problems and could bring your entire app down. Let's say you have a traffic of 100 requests a second. With our 3-second query example, if you get 100 requests in one second and your cache is cold, you'll end up with 300 processes all running the same uncached query.
- A solution is **cache warming**. Cache warming is when websites artificially fill the cache so that real visitors will always get a cache hit. Essentially, sites that engage in cache warming are preparing the cache for visitors, rather than allowing the first visitor to get a cache miss. This ensures every visitor has the same experience.
- Warm cache maintains useful data that your system requires. It helps you achieve faster processing. It will be time-consuming if for each request the process has to query a DB to get this information. so it would be a good idea to cache it; and that would be feasible through warm cache. This cache should be maintained regularly otherwise your cache may grow in size with unnecessary data and you might notice performance degradation.
- I.e.
  Update the cache instead of clearing it after an insert.
  Furthermore, you warm the cache when you start the server.

Scaling The Backend:

- **Load Balancing:**
- We can use a load balancer to distribute the weight. A **load balancer** acts as the "traffic cop" sitting in front of your servers and routing client requests across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization and ensures that no one server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts to send requests to it.

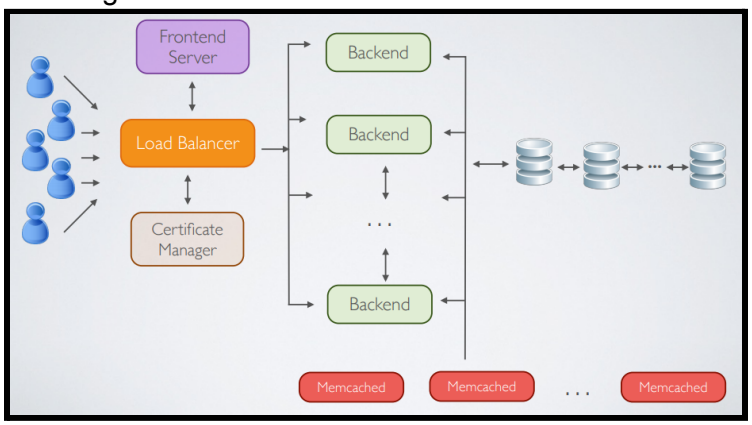


This is a bad design because each memcached in each backend will contain the same information. Your duplicating information over and over again.

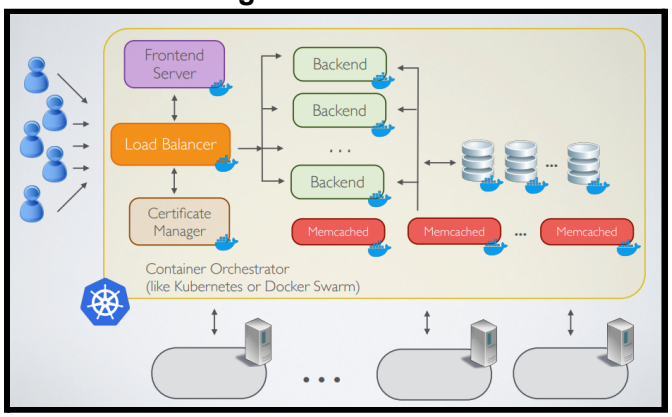


Memcached is a distributed shared cache. This means that while there are many memcached servers, in reality, we're dealing with a unified memcached.

- **Database Sharding:**
- **Database sharding** is the process of breaking up large tables into smaller chunks called shards that are spread across multiple servers. The idea is to distribute data that can't fit on a single node onto a cluster of database nodes.

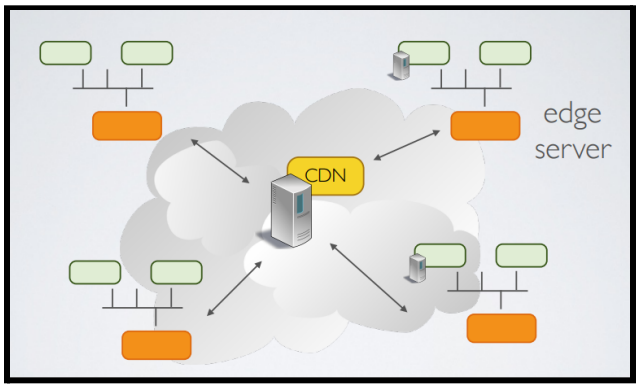


- **Automatic Scaling with container Orchestration:**



At certain times, our web application will have more users using it and at other times, our web application will have few or no users using it. For example, for Amazon, its 2 busiest days are Black Friday and Christmas. Furthermore, consider an Amazon-like website but only for Canada. At night, it will have few people using it. We can use Kubernetes to automatically scale our website based on the stress/load.

- **CDN (Content Distribution Network):**



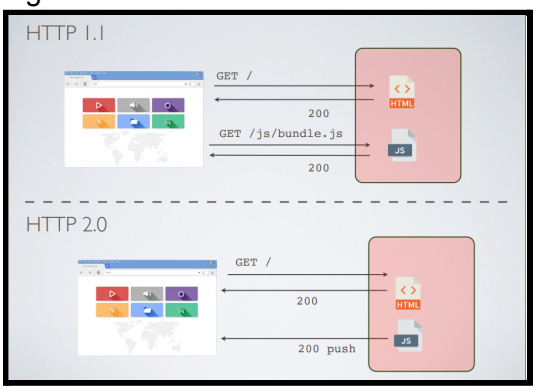
- A **content delivery network (CDN)** refers to a geographically distributed group of servers which work together to provide fast delivery of Internet content.
- A CDN allows for the quick transfer of assets needed for loading Internet content including HTML pages, javascript files, stylesheets, images, and videos. The popularity of CDN services continues to grow, and today the majority of web traffic is served through CDNs, including traffic from major sites like Facebook, Netflix, and Amazon.
- A CDN is a highly-distributed platform of servers that helps to minimize delays in loading web page content by reducing the physical distance between the server and the user. This helps users around the world view the same high-quality content without slow loading times.
- Without a CDN, content origin servers must respond to every single end user request. This results in significant traffic to the origin and subsequent load, thereby increasing the chances for origin failure if the traffic spikes are exceedingly high or if the load is persistent.
- By responding to end user requests in place of the origin and in closer physical and network proximity to the end user, a CDN offloads traffic from content servers and improves the web experience, thus benefiting both the content provider and its end users.

Frontend Packing:

- If your frontend has a lot of HTML/JS/CSS files, loading the files will take a long time.
- A solution is to use webpack.
- Webpack's mission is to take many different modules, files and assets and bundle them together. It transfers their content into a single file or a smaller group of files. It also manages all complexities regarding your dependencies, making sure that code is loaded in the correct order.

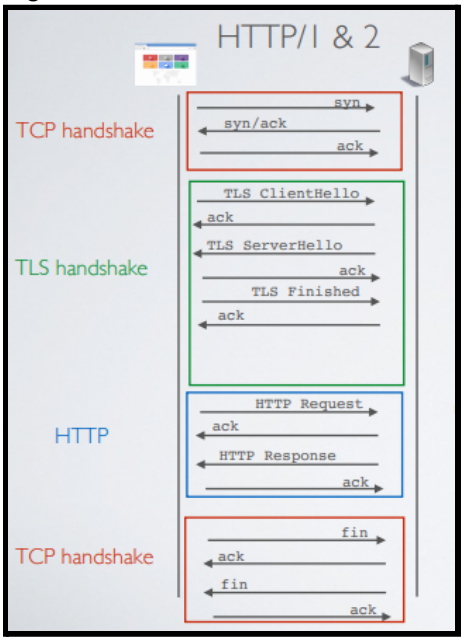
HTTP Versions 2.0 and 3.0:

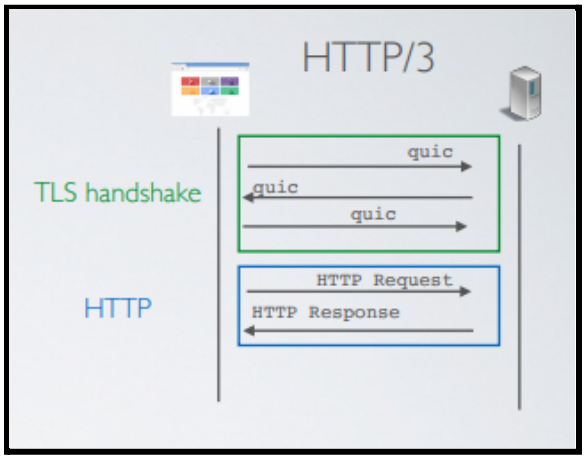
- **HTTP 2:**
- HTTP/2 enables multiplexing which means sending multiple HTTP responses for a given request (aka push).
- E.g.



With HTTP 1.1, you had to make GET requests separately for the HTML, JS, and CSS files. With HTTP 2.0, if a user makes a GET request for the HTML page, the JS and CSS files are also sent back with the HTML file.

- It was proposed by Google and was originally called SPDY.
- It was adopted as a standard in 2015 (RFC 7540).
- HTTP/2 is compatible with HTTP/1 (same protocols).
- **HTTP/3:**
- Is still a work in progress. However, Chrome is compatible with HTTP 3.
- HTTP/3 is compatible with HTTP/2 and HTTP/1.
- The main difference between HTTP/3 and HTTP/2 is that HTTP/3 uses the UDP protocol instead of the TCP protocol. UDP is a lot faster than TCP.
- E.g.





Short Polling vs Long Polling:

- **Short Polling:**
- **Short polling** is a technique where the frontend requests an update from the backend every few seconds and the backend replies right away regardless if there is an update or not. In this case, many requests/responses are wasted.
  I.e. Send a request to the server and get an instant answer. Do this every few seconds, minutes, etc to keep your application up-to-date. But, this costs a lot of requests.
- Twitter uses short polling.
- **Long Polling:**
- **Long polling** is a technique where the frontend requests an update from the backend and waits for the response and the backend replies to the update request only when there is an update. In this case, no requests/responses are wasted and updates are processed as soon as they arrive.
  I.e. Long polling is technique where the server elects to hold a client connection open for as long as possible, delivering a response only after data becomes available or timeout threshold has been reached. After receiving a response, the client immediately sends the next request.
- Facebook uses long polling.

WebSockets:

- A **WebSocket** is a persistent connection between a client and server. WebSockets provide a bidirectional, full-duplex communications channel that operates over HTTP through a single TCP/IP socket connection. **Note:** WebSockets do not rely on HTTP at all except for initialization.
- A full-duplex system allows communication in both directions to happen simultaneously. E.g. Land-line telephone networks are full-duplex since they allow both callers to speak and be heard at the same time.
- WebSockets are similar to low-level POSIX sockets.

WebRTC (Web Real-Time Communications):

- It is a full-duplex communication between clients (browsers).
- It provides peer-to-peer (P2P) communications, which is perfect for sending text, video, audio without going through the server except for initialization and signalling that goes through the server usually using WebSockets.
- WebRTC has no signaling of its own and this is necessary in order to open a WebRTC peer connection. WebRTC can achieve this by using other transport protocols such as HTTPS or secure WebSockets.
  I.e. To connect a WebRTC data channel you first need to signal the connection between the two browsers. To do that, you need them to communicate through a web server in some way. This is achieved by using a secure WebSocket or HTTPS.
- The main difference between WebSockets and WebRTC is that WebSockets are meant to enable bidirectional communication between a browser and a web server while WebRTC is meant to offer real time communication between browsers.

Progressive Web Applications (PWA):

- PWAs are web applications that can be installed on your system.
- It works offline when you don't have an internet connection, leveraging data cached during your last interactions with the app.
- It relies on the browser's local storage to store the frontend and checks for updates with the server.
- It relies on web-workers for caching and communication.

i18n & L10n:

- **Internationalization (i18n)** is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes.
- I.e. The purpose is to make an application language agnostic.
- **Localization (L10n)** is the process of adapting internationalized software for a specific region or language by adding locale specific components and translating text.
- I.e. The purpose is to adapt an application for a specific language (locale).
- Internationalization is the process of designing and developing your software so it can be adapted and localized to different cultures, regions, and languages while localization is the adaptation of your software to meet the language, culture, and other requirements of each locale.
- Internationalization helps you build your software t with future markets and languages in mind. It's the process of neutralizing the code, content, and design so that, down the road, it'll be easier to adapt your product to additional cultures without having to completely re-engineer it. Localization typically follows internationalization.
- **Note:** Localization is not only about language translation.
  We also have to deal with the following:
  - Number format
  - Date/Time
  - Punctuation
  - Sort orders
  - Units and conversion
  - Currency
  - Paper size
  - Page layout
- You can configure your locale preference in your browser.
- The language that the website is in can be found in the request header.
  E.g.

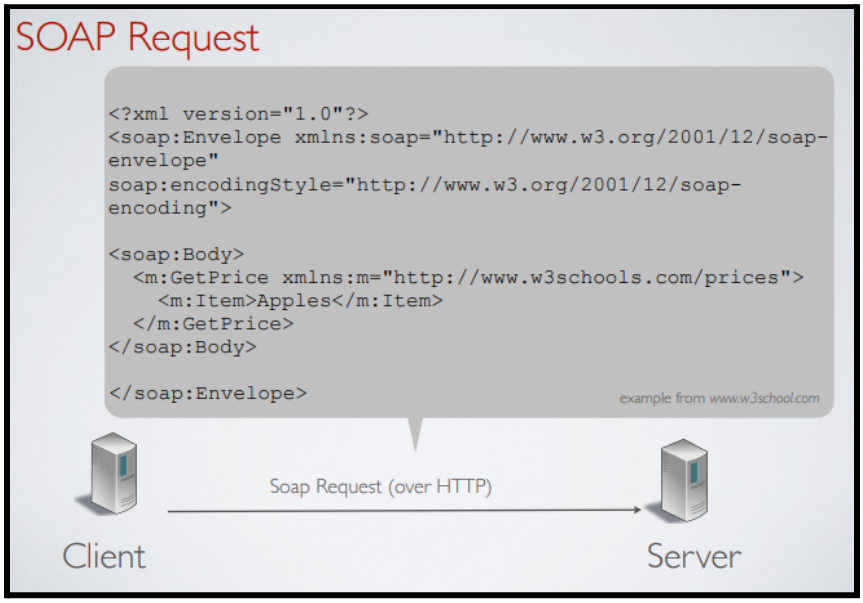
- Some ways to make your websites language agnostic:
  - Store the language preference in the URL.
  - Store the language preference in the user's profile.
  - Store the language preference in a cookie.

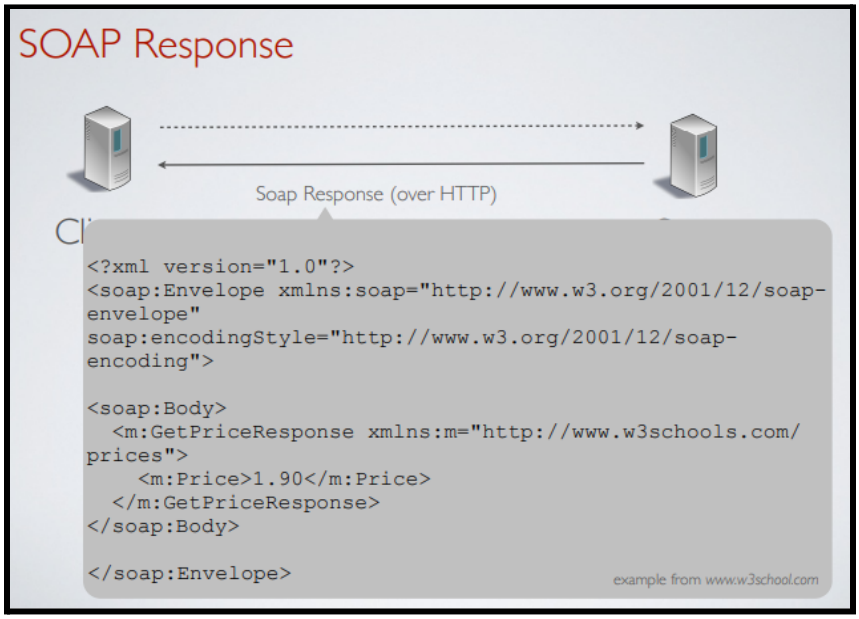
Web Services:

- Is the predecessor to web APIs.
- A **web service** is a standardized medium to propagate communication between the client and server applications on the internet. Web services are mostly used between web servers (B2B).
- A client would invoke a series of web service calls via requests to a server which would host the actual web service. These requests are made through what is known as Remote Procedure Calls (RPC). RPCs are calls made to methods which are hosted by the relevant web service. The requests are typically made through HTTP but can also be made through SMTP. The main component of a web service design is the data which is transferred between the client and the server using XML. This provides a common platform for applications developed in various programming languages to talk to each

other. Web services use SOAP for sending the XML data between applications. This data is called a SOAP message, which is an XML document.

- **SOAP (Simple Object Access Protocol)** is known as a transport-independent messaging protocol. SOAP is based on transferring XML data called SOAP Messages. Each message contains an XML document. Only the structure of the XML document follows a specific pattern, not the content.
  Here is what a SOAP message consists of:
  - Each SOAP document needs to have a root element known as the <Envelope> element. The root element is the first element in an XML document.
  - The "envelope" is in turn divided into 2 parts. The first is the header, and the next is the body.
  - The header contains the routing data which is basically the information which tells the XML document to which client it needs to be sent to.
  - The body will contain the actual message.



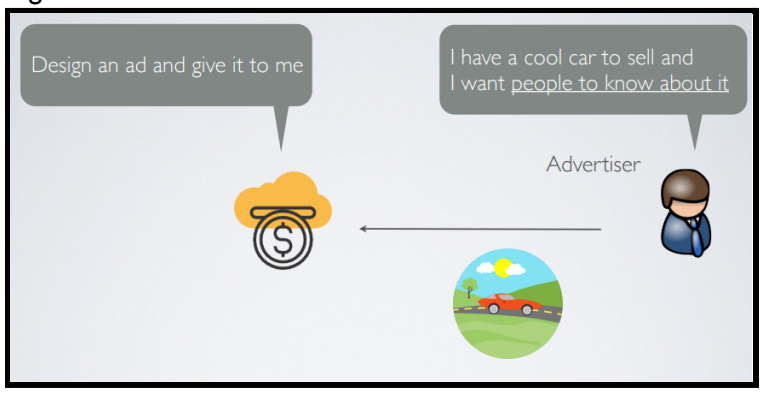


- The **WSDL (Web services description language)** file is an XML-based file which basically tells the client application what the web service does.
  A web service cannot be used if it cannot be found. The client invoking the web service should know where the web service actually resides.
  Secondly, the client application needs to know what the web service actually does, so that it can invoke the right web service. This is done with the help of the WSDL, known as the Web services description language. By using the WSDL document, the client application would be able to understand where the web service is located and how it can be utilized.
  The WSDL file provides a way to describe your web service.
  E.g.

- **Universal Definition Language (UDDI)** is a standard for describing, publishing, and discovering the web services that are provided by a particular service provider. It provides a specification which helps in hosting the information on web services. It provides a way to advertise your web service.
- While web services are a good idea, they have not been widely adopted because:
  - Web services are very flexible but very complex architecture.
  - Standards for web services evolve faster than development frameworks.
  - Ad-hoc solutions adopted by the main actors of the web.
    I.e. JSON replaced XML as JSON is more human and computer readable.
    Furthermore, web APIs replaced web services.

Advertising, Analytics and Tracking:

- **Advertising:**
- One of the oldest and most popular ways to generate revenue for a webpage.
- E.g.



What's happening is that an advertiser will go to an agency with some ads and ask the agency to find some content publishers to publish their ads on the content publisher's website. The content agency then looks for content publishers to showcase the ads on their website so that visitors of that website can see the ads.
Note: Sometimes, advertisers go directly to the content publisher. Examples of this are Google and Facebook.

- There are 2 main popular models of online advertising:
    1. **Click Banners:**
    - On other people's websites.
    - Examples include:
        - Pay per click
        - Pay per view
        - Pay per transaction
    2. **Sponsored Links:**
    - On the search engine result page.
    - Buying keywords (bidding price).
    - Google does this

- If you embed ads in your webpage/webapp, the ad network rewards you with cash every time a visitor clicks on an ad on your webpage.
- For the web programmer, a javascript snippet to be inserted in the webpage will perform ajax requests to the ad networking company. The ad is shown in an iframe.
- For the visitor, a third party cookie will track his/her visits through different sites to display more relevant ads.
- **Web Scraping and Click Fraud:**
- There are 2 main types of advertising fraud:
    1. **Web Scraping:**
    - **Web Scraping** occurs when a website extracts, collects and aggregates data from other websites to make it seem legit. Then, when users search for certain keywords, search engines will prioritize those websites.
      I.e. They are spamming search engines (spamdexing).
      The goal is to attract visitors to your website and fool them to click on ads.
    2. **Click Fraud:**
    - **Click fraud** is having a bot (a computer program) that automatically clicks on:
        - ads displayed on your website to increase your earnings or
        - ads anywhere on the web but targeting specific ads to increase the expenses of your competitors.
- **Web Analytics:**
- Is used to maximize your revenue from advertisement.
- It looks at the following information:
    - Which website guides the users to your website?
    - What are the keywords that they typed in the search engine that guide them to your website?
    - What do they do on your website?
    - How long do they stay?
    - What pages do they look at?
    - Where are they from geographically?
    - What are their hobbies?
    - How old are they?
- There are 2 techniques for web analytics:
    1. **Log file analysis:**
    - Is done on the server side.
    - Server side code analyzes the web server logs.
    - Can collect IP addresses, rate of requests, etc.
    2. **Page tagging analysis:**
    - Is done on the client side.
    - Javascript code analyzes the user's interactions.
    - Can collect the type of device the user is on (laptop/cell phone/desktop) and the brand of the device (Apple/Samsung/Google)
- You can do web analytics on your own by mixing log analysis and page tagging or you can ask another company, like Google, to do it for you. If you do web analytics as a service, you can do page tagging only.
- **Web Tracking:**
- **Third-party cookies** are cookies with a unique ID to identify the same user visiting different websites. Google and other websites place third-party cookies in their websites or ads to learn more information about their users.

- **Browser fingerprinting** refers to tracking techniques that websites use to collect information about you. Some information collected include:
    - the User agent header
    - the Accept header
    - the Connection header
    - the Encoding header
    - the Language header
    - the list of plugins
    - the platform
    - the cookies preferences (allowed or not)
    - the Do Not Track preferences
    - the timezone
    - the screen resolution and its color depth
    - the use of local storage
    - the use of session storage
    - the presence of AdBlock
    - the list of fonts
- Browsing in **privacy mode** disables the browser data storage.
  I.e. It disables the following:
    - (frontend) web cache
    - HTTP cookies
    - HTML5 local storage
    - Flash/Silverlight cookies

  However, it does not protect against browser extensions.
- **Do Not Track** is a HTTP header field that was proposed in 2009. If you have this on, the browser will tell the website to not to use third party cookies. This is completely useless because websites can decide whether or not to honor such a request.
  I.e. The browser asks the website to not use third party cookies, but it is up to the website to decide if they want to comply or not.