React Context Api

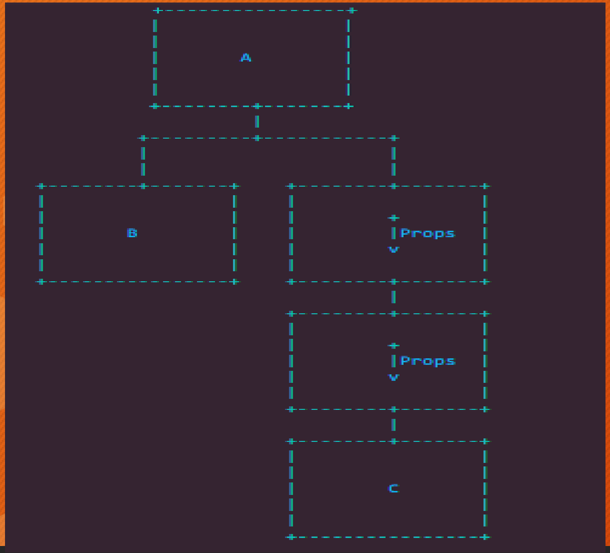
Advanced React Course

Why Context API

Single Component or Multiple Components

Single Component or Multiple Components?

- Every state change results in a re-render of the entire application
- Code sharing/reusability Not Supported
- Testing would be 100% integration
- Working together on the codebase with multiple engineers would just be terrible
- State would be a challenge: Knowing which pieces of state and event handlers went with what parts of JSX



What is prop drilling?

- Prop-drilling refers to the technique of passing down variables to sub components
- Prop drilling is also called "threading" refers to the process react developers   go through to get data to parts of the React Component tree

What problems can prop drilling cause?

- As an application grows, drilling through many layers of components to pass data from parent component to child component.
- Here are various other situations where prop drilling can cause some real pain
  - Refactor the shape of some data
  - Over-forwarding props (passing more props than is necessary) due to (re)moving a component that required some props but they're no longer needed.
  - Renaming props halfway through (ie <A on={this.state.on} /> renders <B IsOn={on} />) making keeping track of property synonyms is difficult.

How can we avoid problems with prop drilling?

- Global Data
- Single Giant Component
- Good Practices for Application State Management
  - React Context Api

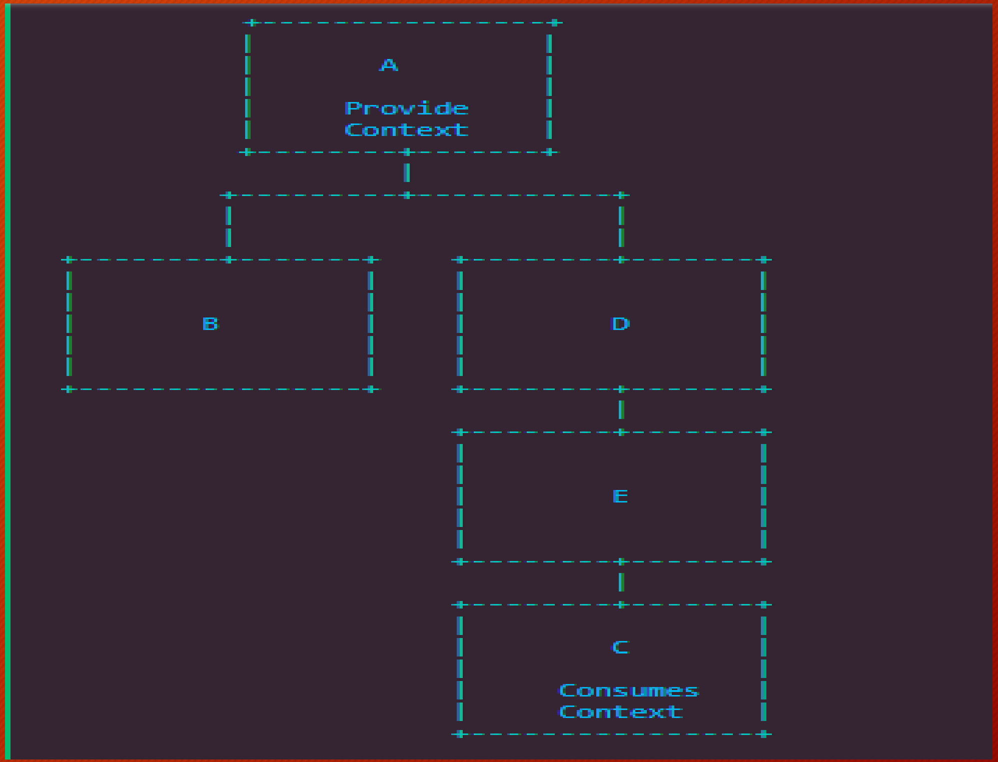
React Context API

React version prior 16.3

React 16.3 Abovr

React Context API Prior 16.3

- React context object pass data implicitly down to each component
- The React context object traverses invisible down the component tree. When a component needs access to the context object, it can access it
- “Props Hiding” from pass-through components



What are use cases for React Context?

How to fetch configuration once from your server, but afterward we want to make this implicitly accessible for all components

React Provider Pattern

This pattern makes the properties accessible in the context and another part where components consume the context.

Lots of React libraries need to make their data pass through all your component tree

Redux needs to pass its store

React Router needs to pass the current location

provider pattern explains how the state layer is glued to your view layer when using a state management library

When using Redux or MobX, the state (store) is passed to your Provider component as props. The Provider component wraps the store into React's context and thus all child components can access the store

Introducing React Hooks

React Hook introduce state management and side-effects in function components

React Hooks enable us to write React applications with only function components without refactoring in to Class Components

Every hook has its own side-effect with a setup and clean up phase

Why React Hooks

Unnecessary Component Refactorings

Side-effect Logic

Wrapper Hell

JavaScript Class Confusion

Unnecessary Component Refactoring's

- React function components are more lightweight (and elegant), people already used plenty of function components
- Refactoring components from React function components to React class components every time state or lifecycle methods were needed
- With Hooks there is no need for this refactoring. Side-effects and state are available in React function components

Side-effect Logic

A side-effect could be fetching data in React or interacting with the Browser API

Every side-effect requires proper setup and clean up phase

In React side-effects were mostly introduced in lifecycle methods (e.g. componentDidMount, componentDidUpdate, componentWillUnmount)

In Class Components all side-effects would be grouped by lifecycle method but not by side-effect

In React Hooks - every hook has its own side-effect with a setup and clean up phas

Wrapper Hell in React

There are dozens of wrapped components due to Render Prop Components (including Consumer components from Reacts Context) and Higher-Order Components. It becomes an unreadable component tree

The actual visible components are hard to track down in the browser's DOM

The rendered tree becomes an unreadable component tree

In React Hooks - All side-effects are sitting directly in the component without introducing other components as container for business logic. The container disappears and the logic just sits in React Hooks that are only functions

Wrapper Hell in React

There are dozens of wrapped components due to Render Prop Components (including Consumer components from Reacts Context) and Higher-Order Components. It becomes an unreadable component tree

The actual visible components are hard to track down in the browser's DOM

The rendered tree becomes an unreadable component tree

In React Hooks - All side-effects are sitting directly in the component without introducing other components as container for business logic. The container disappears and the logic just sits in React Hooks that are only functions

JavaScript Class Confusion

one of the hypotheses of introducing the Hooks API is a smoother learning curve for React beginners when writing their React components without JavaScript classes in the first place.

Building Blocks Of react -hooks

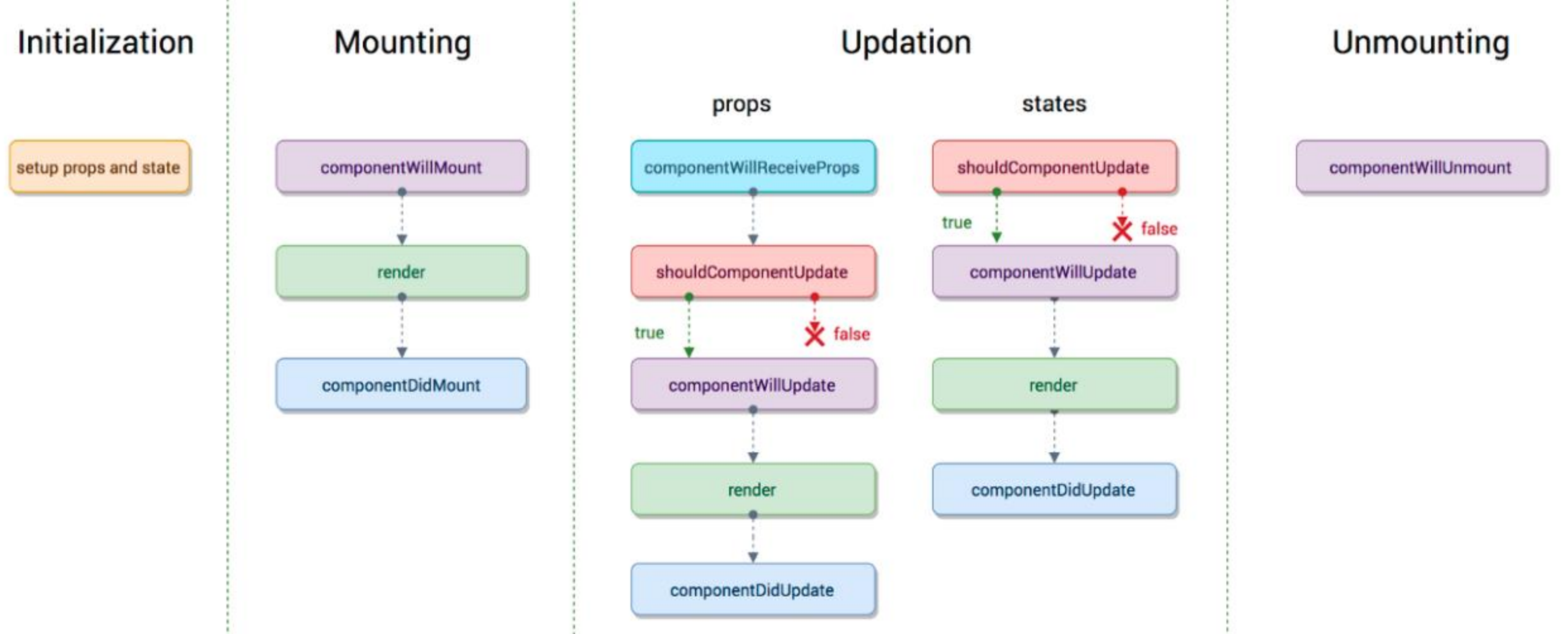
useState

useEffect

The Rules of Hooks

Don’t call Hooks inside loops, conditions, or nested functions

Only Call Hooks from React Functions



Initialization
Mounting
Updation
Unmounting
setup props and state
componentWillMount
render
componentDidMount
props
componentWillReceiveProps
shouldComponentUpdate
true
false
componentWillUpdate
render
componentDidUpdate
states
shouldComponentUpdate
true
false
componentWillUpdate
render
componentDidUpdate
componentWillUnmount

What is PWA

Imagine that you had the ability to build a website that worked completely offline, offered your users near instant load times, and yet was secure and resilient to unreliable networks at the same time

PWA Characteristics

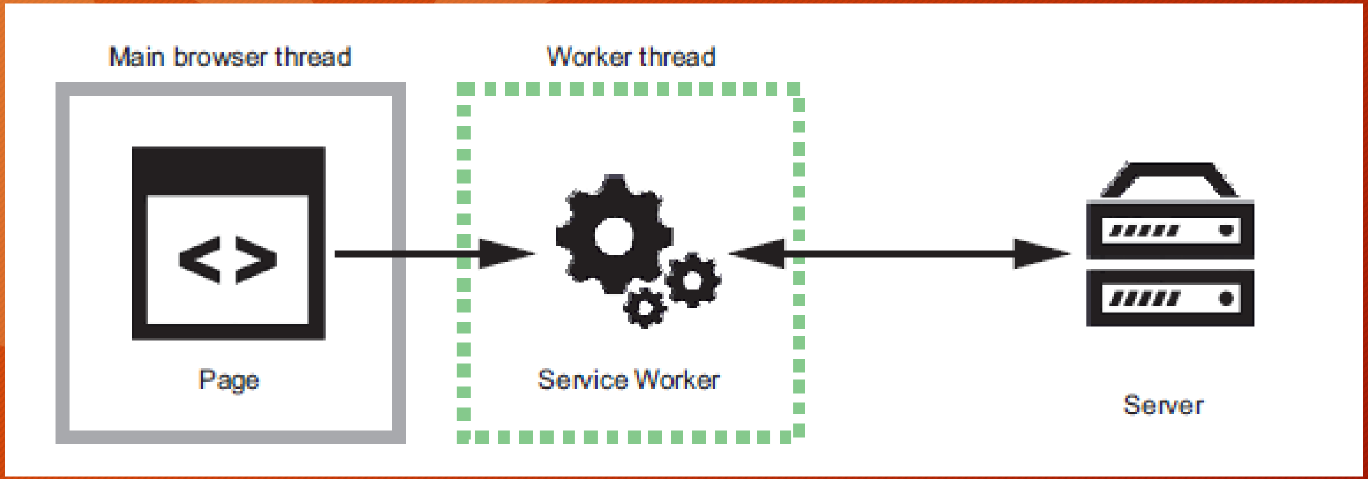
- Responsive - It adjusts to smaller screen sizes.
- Connectivity-independent - It works offline due to Service Worker caching
- Interactive with a feel like a native app’s - It’s built using the Application Shell Architecture
- Always up-to-date - Service Worker update processg
- Safe - It works over HTTPS.
- Discoverable - A search engine can find it
- Installable - It’s installable using the manifest file
- Linkable - It can be easily shared by URL

Building Blocks PWA

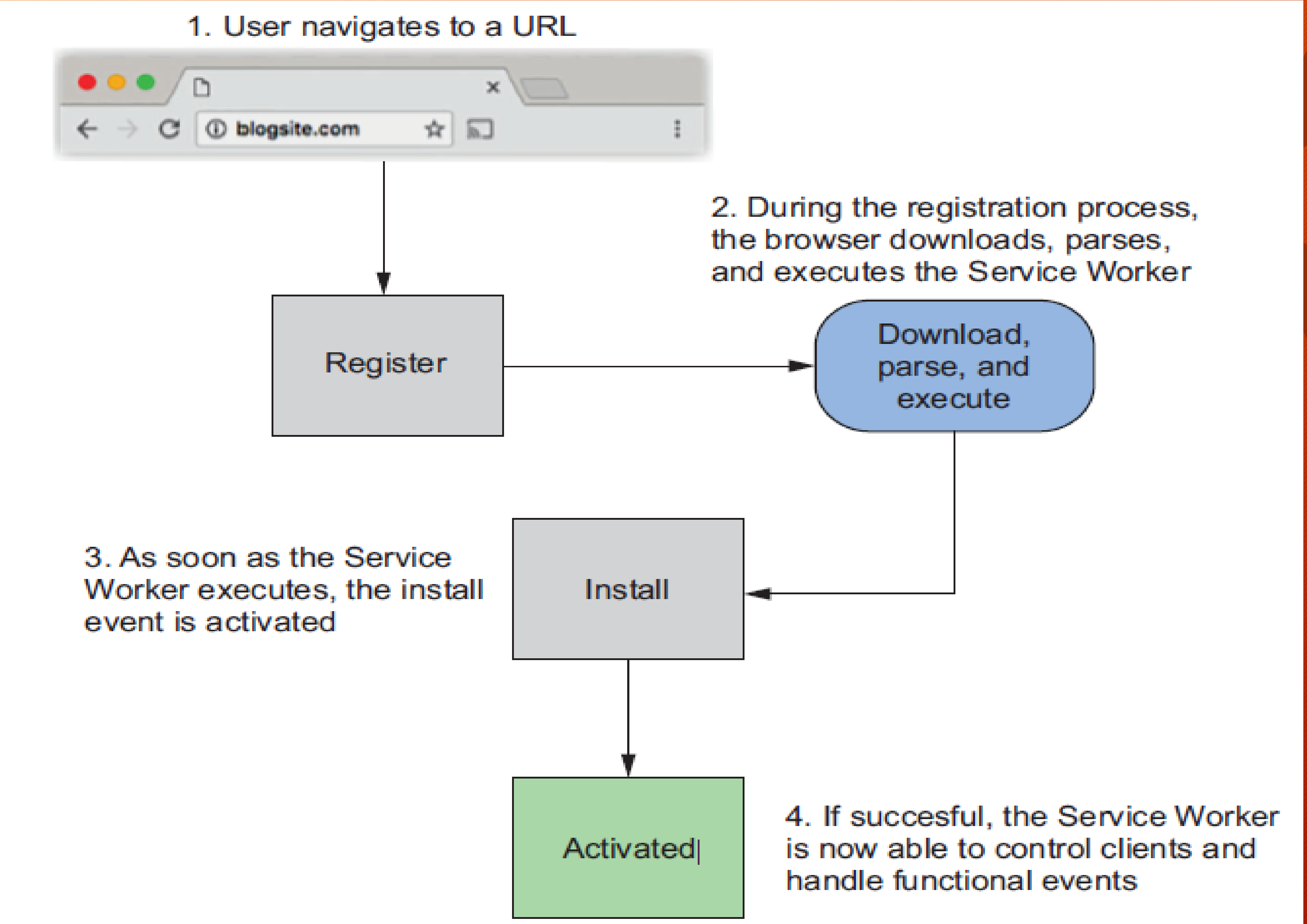
- Manifest file
  - contains information about the website, including its icons, background screen, colors, and default orientation
- ServiceWorker
  - Tap into network requests and build better web experience

ServiceWorker -*The key to PWAs*

- Service Workers are worker scripts that run in the background
- Runs in its own global script context
- Isn’t tied to a specific web page
- Can’t modify elements in the web page—it has no DOM access
- Is HTTPS only



SERVICE WORKER LIFECYCLE


1. User navigates to a URL
blogsite.com
2. During the registration process, the browser downloads, parses, and executes the Service Worker
Register
Download, parse, and execute
3. As soon as the Service Worker executes, the install event is activated
Install
Activated|
4. If succesful, the Service Worker is now able to control clients and handle functional events

Service Worker Registration

```
<html>
<head>
<title>The best website ever</title>
</head>
<body>
<script>
// Register the service worker
if ('serviceWorker' in navigator) {
     navigator.serviceWorker.register('/sw.js').then(function(registration) {
    // Registration was successful
    console.log('ServiceWorker registration successful with scope: ',
     registration.scope);
}).catch(function(err) {
    // registration failed :(
    console.log('ServiceWorker registration failed: ', err);
    });
}
</script>
</body>
</html>
```

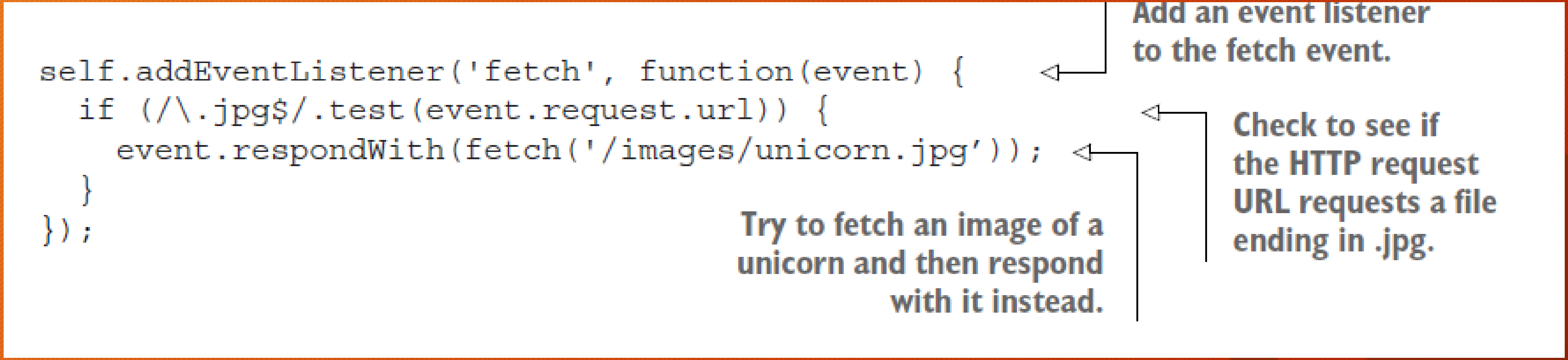
Check to see if the current browser supports Service Workers.

If it does, register a Service Worker file called sw.js.

Log to the console if successful.

If something goes wrong, catch the error and log to the console.

“



```
self.addEventListener('fetch', function(event) {
  if (/\.jpg$/.test(event.request.url)) {
    event.respondWith(fetch('/images/unicorn.jpg'));
  }
});
```

Add an event listener to the fetch event.
Check to see if the HTTP request URL requests a file ending in .jpg.
Try to fetch an image of a unicorn and then respond with it instead.

ServiceWorker - *Code*

Progressive Enhancement?

- Progressive enhancement means application gets progressively better as user’s internet connection improves
- Application is thus an application adaptive to the connection.
- The progressive enhancement paradigms urge us to reorganize our site's content so that the importance stuff loads as quickly as possible, and then the bells and whistles come in

The RAIL model

- Google calls a "user-centric performance model
- It's a set of guidelines for measuring our app's performance
- Use RAIL's principles to speed up our application and ensure that it performs well enough for all users.
- RAIL outlines four specific periods in an application's life cycle
  - Response
  - Animation
  - Idle
  - Load

RAIL Model

- Load
  - RAIL says that the optimal load time is one second (or less)
  - Doesn't mean your entire application loads in one second.
  - The user sees content within one second. They get some sense that the current task (loading the page) is progressing in a meaningful way
- **Idle**
  - Once your application is done loading, it is idle (and also will be idle between actions) until a user performs an action.
  - RAIL argues that we should use this time to continue loading in parts of the application.
  - If Application initial load is just the bare bones version of our app, let load the rest (progressive enhancement!) during idle time.

RAIL MODEL

- **Animation**
  - Users will note a lag in animations if they are not performed at 60 fps.
  - RAIL also defines scrolling and touch gestures as animations
  - Even if applications have no animations, if scroll is laggy, you have a problem, will negatively affect the perceived performance
- **Response**
  - When user performs an action ,Applications have 100 ms to provide a response that acknowledges their action
  - Note that some actions will take longer to complete, Application don’t need to complete the action in 100 ms, but Application do have to provide some response
  - *Meggin Kearney* puts it, "the connection between action and reaction is broken. Users will notice

Timeline- References

- >16ms: Time per frame of any animation/scrolling.
- >100ms: Response to user action.
- >1000ms: Display content on the web page.
- 1000ms+: User loses focus.
- 10,000ms+: User will likely abandon the page.

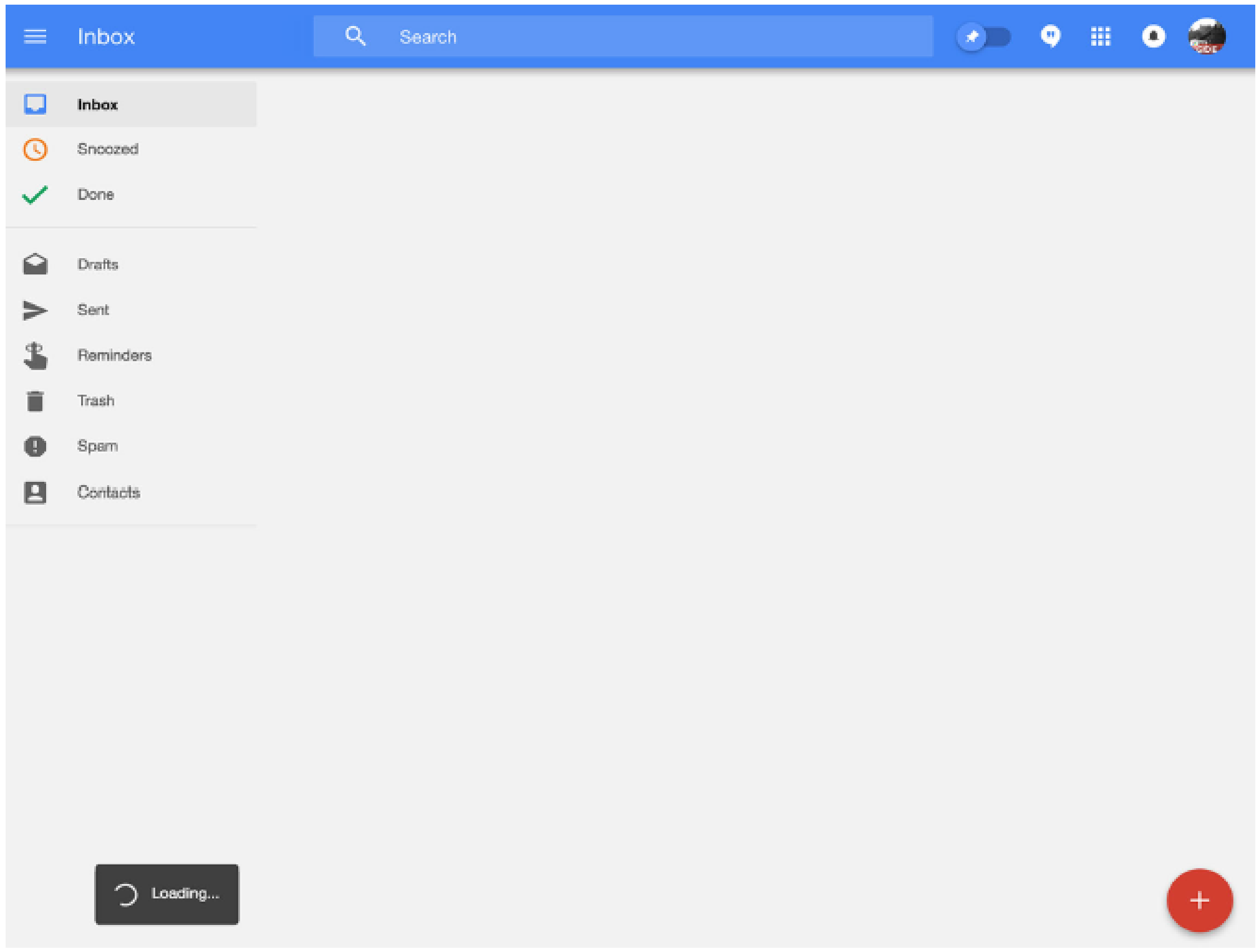
If your application performs according to these specifications, you're in good standing

Measuring using the timeline

- Google Chrome Dev tools
- **PageSpeed Insights - https://developers.google.com/speed/pagespeed/insights/**

AppShell Pattern

- If you have an existing web application use handy tool called Lighthouse to evaluate application and get useful performance and audit information
- ***Front-end architectural approaches to building PWAs***
  - ***The Application Shell Architecture***
    - *UI shell* - minimal HTML, CSS, and JavaScript required to power the user interface
    - Load the UI shell and cache it, you can load the dynamic content into the page later

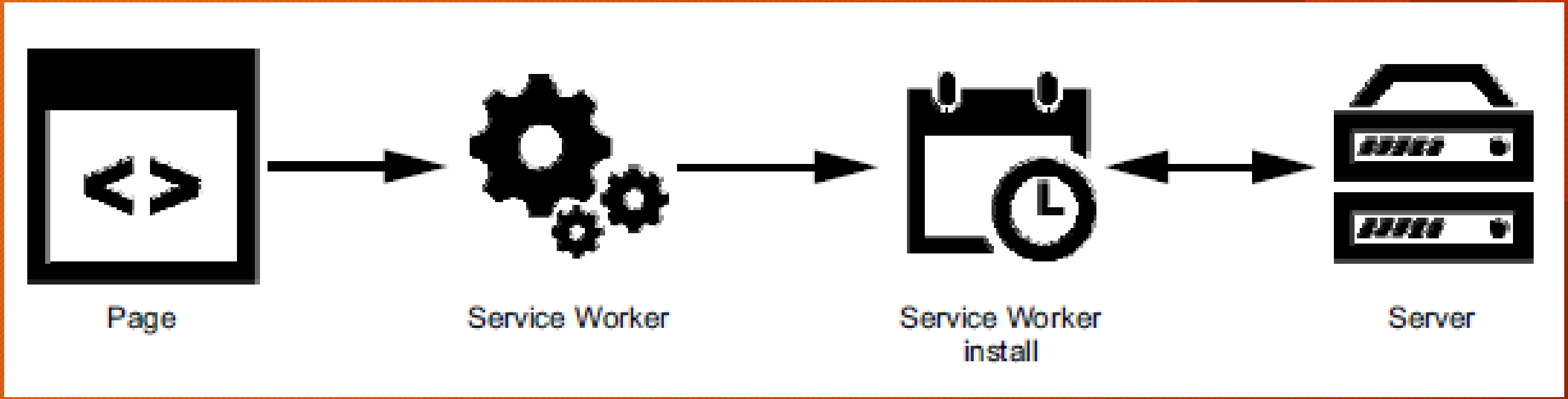


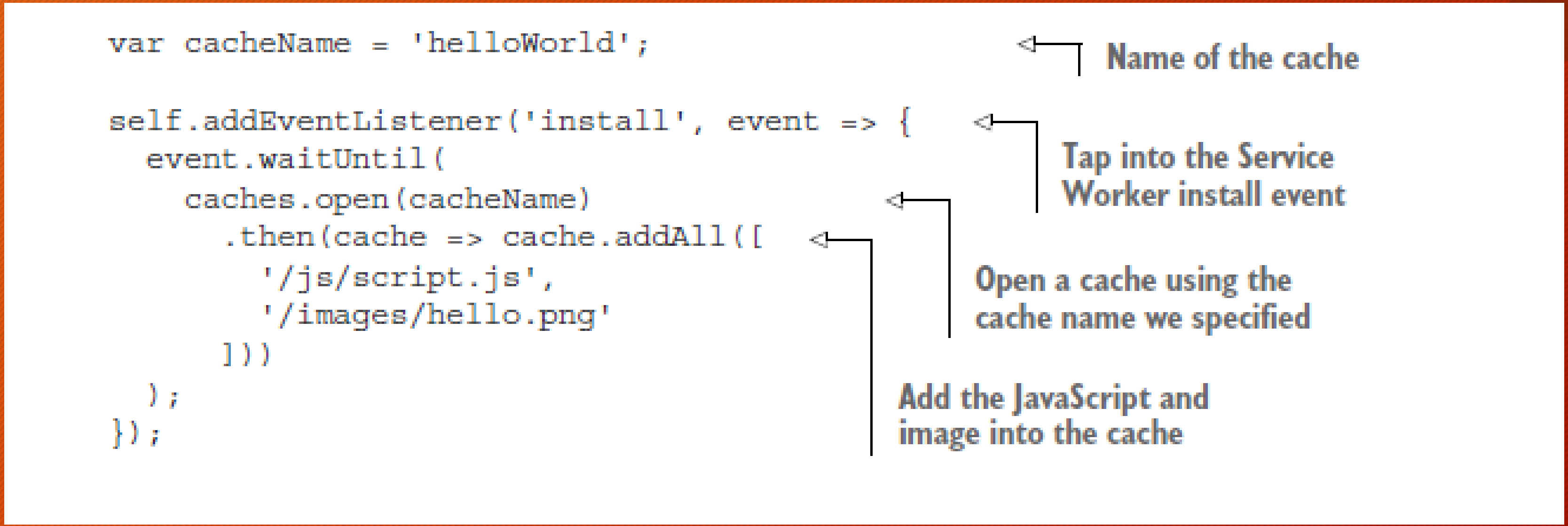
The PRPL pattern

- **PRPL** stands for **Push**, **Render**, **Pre-cache**, **Lazy-load**
- *“Push critical resources for the initial URL route.”*
- render Application initial route
- Precaching means that once those assets are loaded, they will go straight into the cache and, if they're requested again, we load them from the cache
- **lazy-loading**--loading resources that aren't needed immediately but may be needed in the future.
  - code splitting

Caching

- *Precaching during Service Worker installation*





```
var cacheName = 'helloWorld';

self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(cacheName)
      .then(cache => cache.addAll([
        '/js/script.js',
        '/images/hello.png'
      ]))
  );
});
```

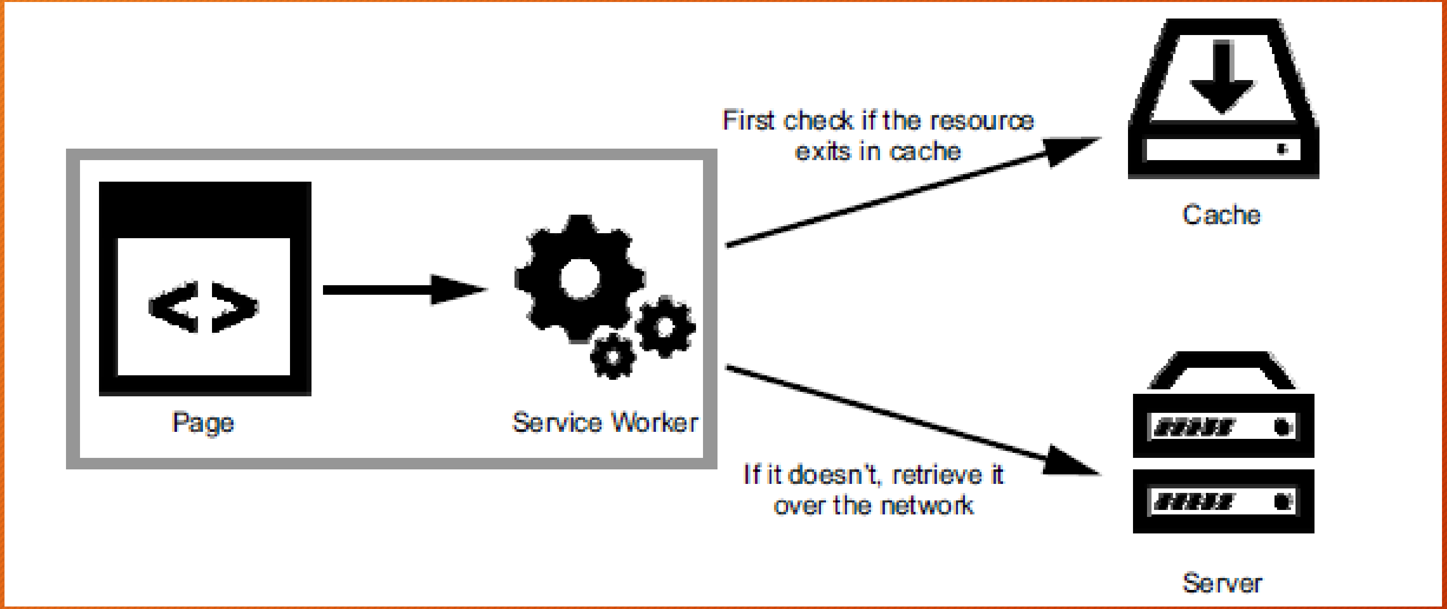
Name of the cache

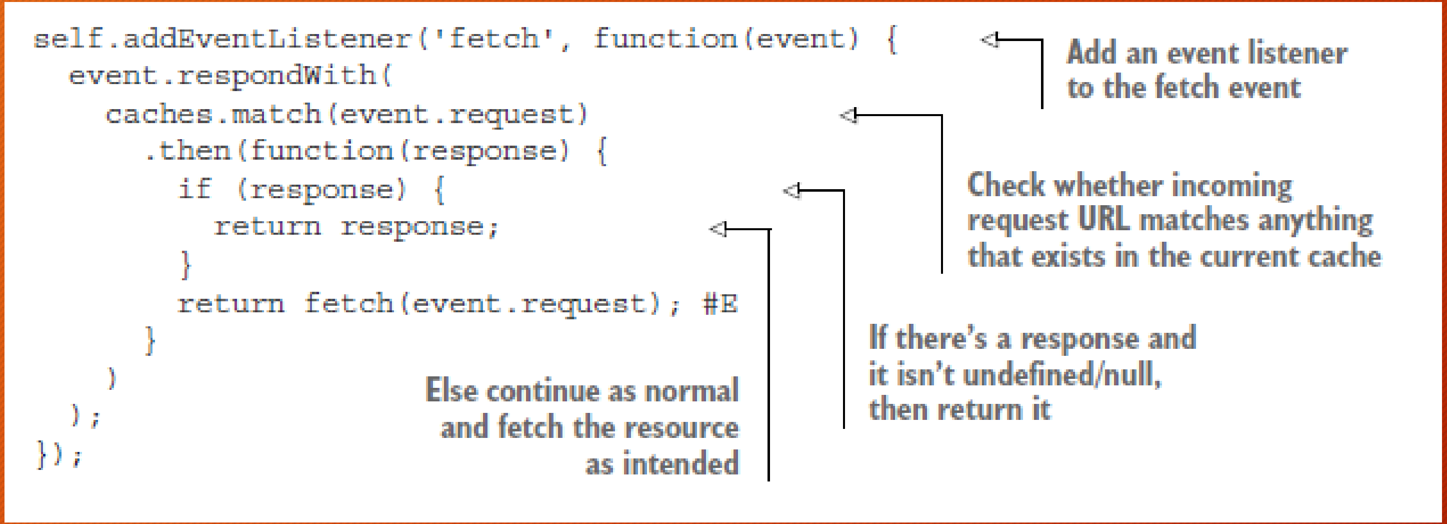
Tap into the Service Worker install event

Open a cache using the cache name we specified

Add the JavaScript and image into the cache

Intercept and Cache


First check if the resource exits in cache
Cache
Page
Service Worker
If it doesn't, retrieve it over the network
Server



```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
    .then(function(response) {
      if (response) {
        return response;
      }
      return fetch(event.request); #E
    }
   )
  );
});
```

Add an event listener to the fetch event

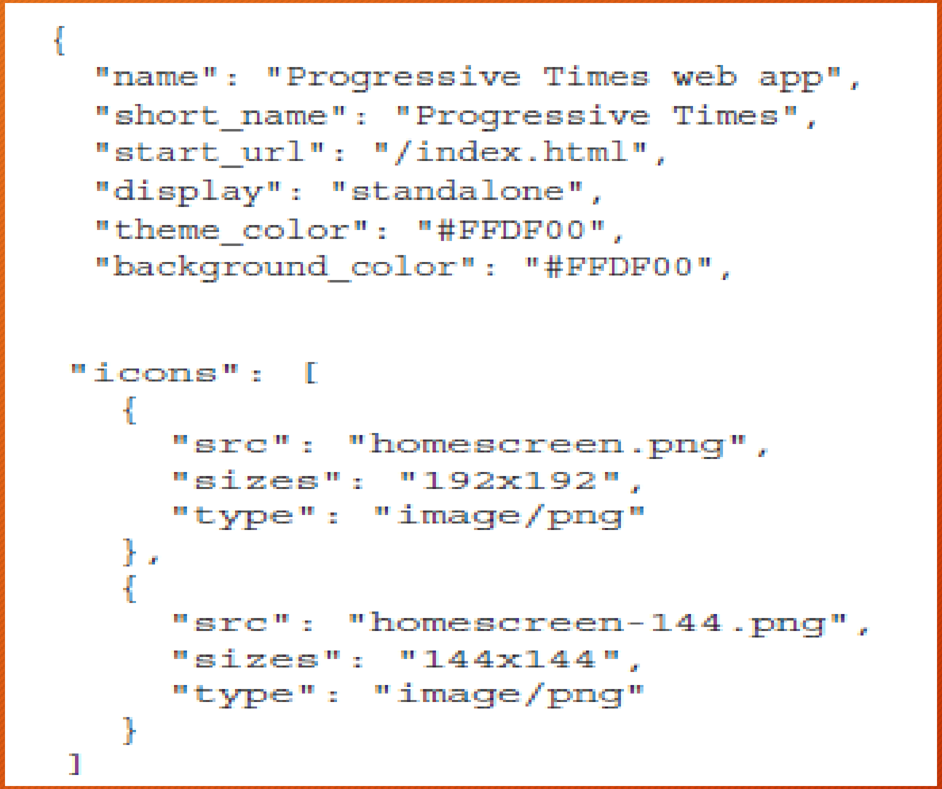
Check whether incoming request URL matches anything that exists in the current cache

If there's a response and it isn't undefined/null, then return it

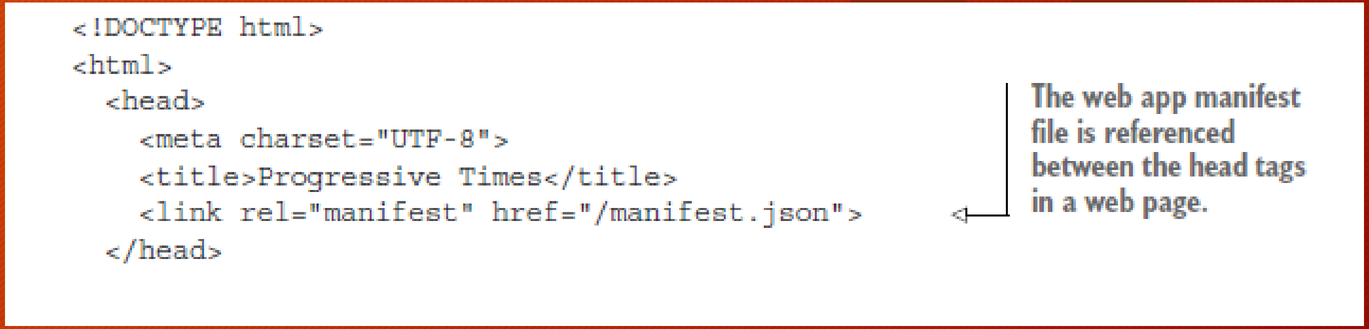
Else continue as normal and fetch the resource as intended

WebApp Manifest file

- JSON file
- web app manifest enables a user to install web applications to
- the home screen of their device and allows you to customize the splash screen, theme colors, and even the URL that’s opened.



```
{
  "name": "Progressive Times web app",
  "short_name": "Progressive Times",
  "start_url": "/index.html",
  "display": "standalone",
  "theme_color": "#FFDF00",
  "background_color": "#FFDF00",


 "icons": [
   {
     "src": "homescreen.png",
     "sizes": "192x192",
     "type": "image/png"
   },
   {
     "src": "homescreen-144.png",
     "sizes": "144x144",
     "type": "image/png"
   }
 ]
```



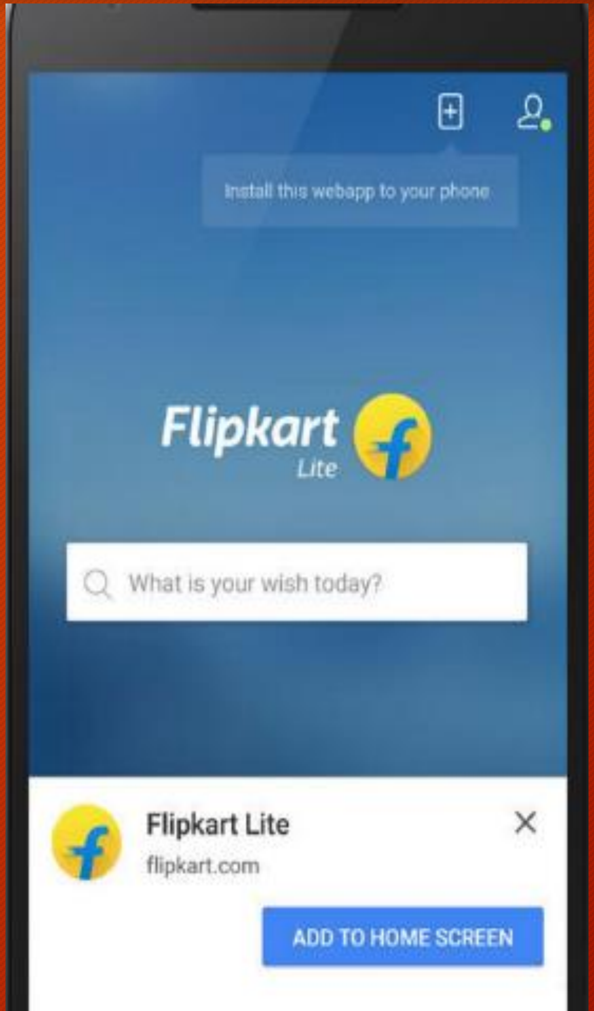
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Progressive Times</title>
    <link rel="manifest" href="/manifest.json">
  </head>
```

The web app manifest file is referenced between the head tags in a web page.

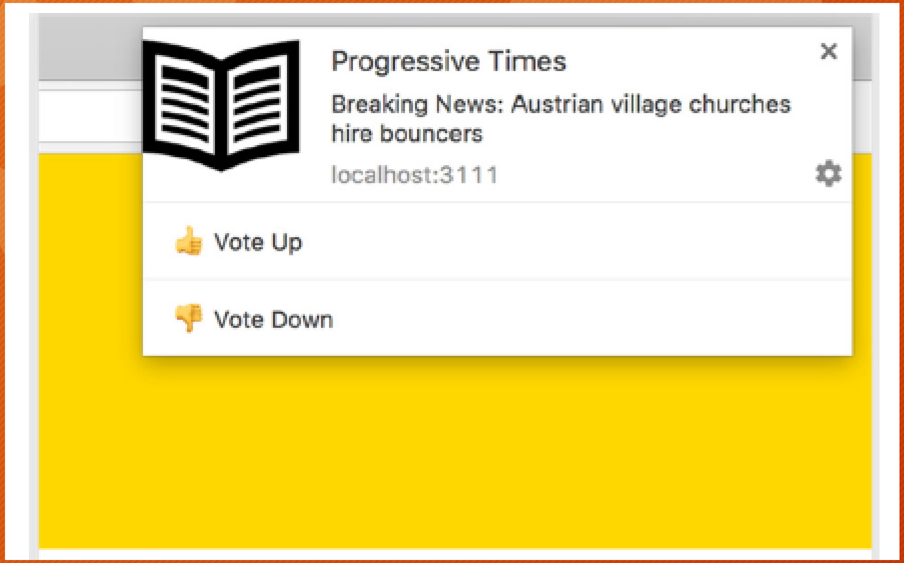
Add to Home Screen

Steps

- You need a manifest.json file.
- Your manifest file needs a start URL.
- You need a 144 x 144 PNG icon.
- Your site must be using a Service Worker running over HTTPS.
- The user must have visited your site at least twice, with at least five minutes between visits

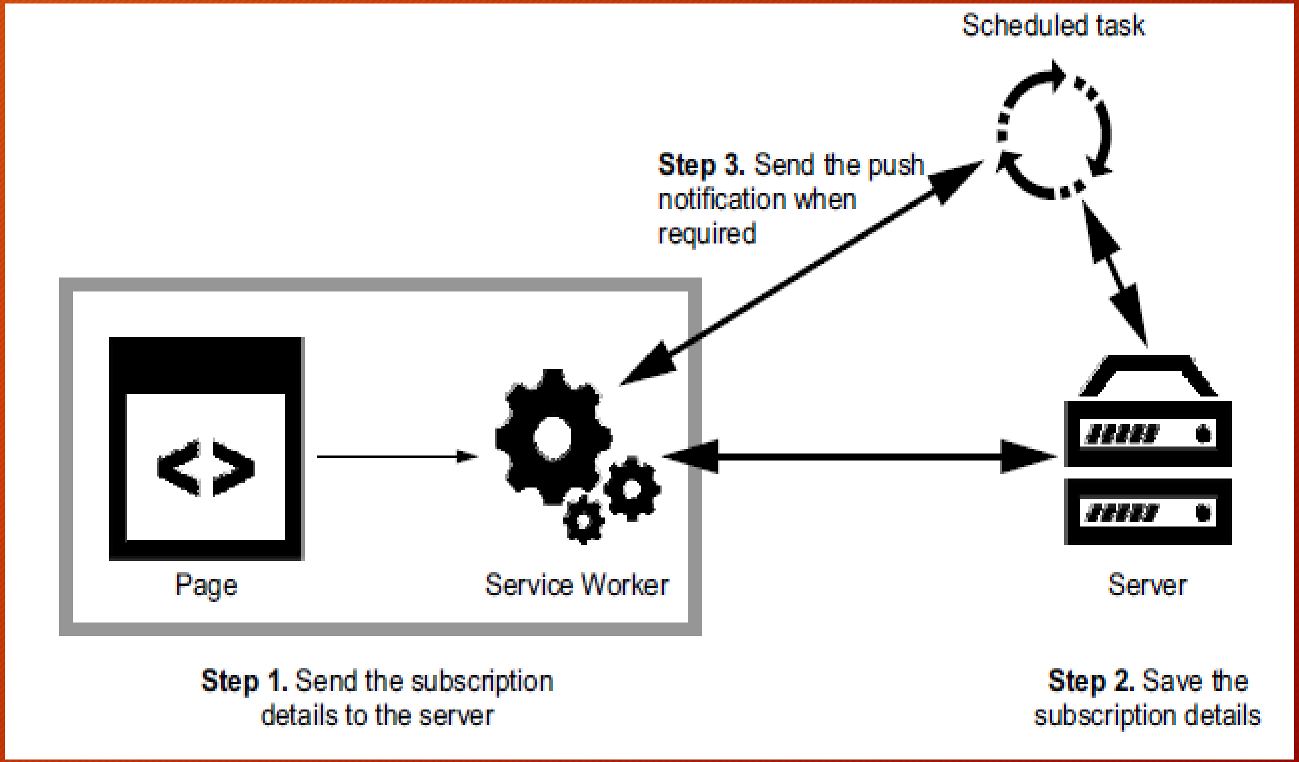


Push Notifications


Progressive Times

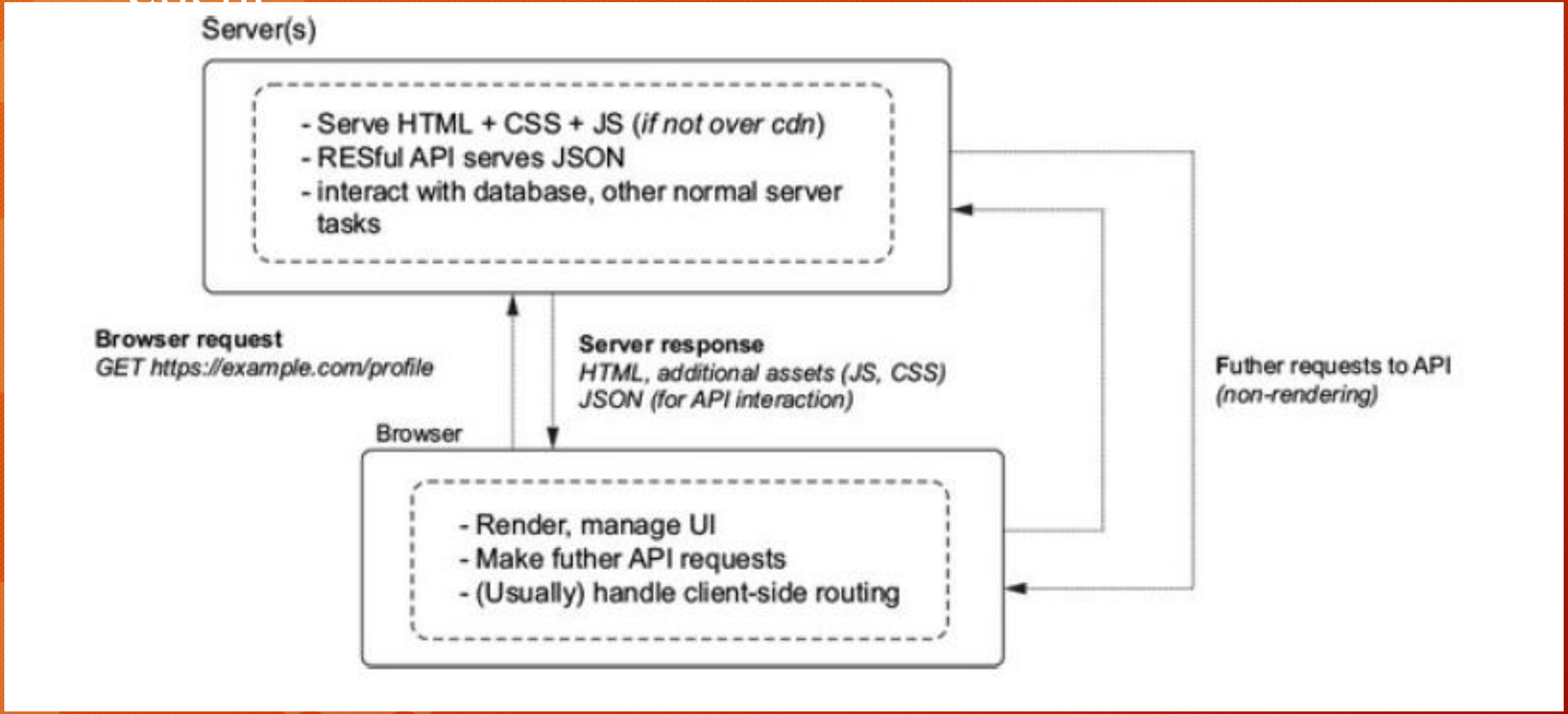
Breaking News: Austrian village churches hire bouncers

localhost:3111

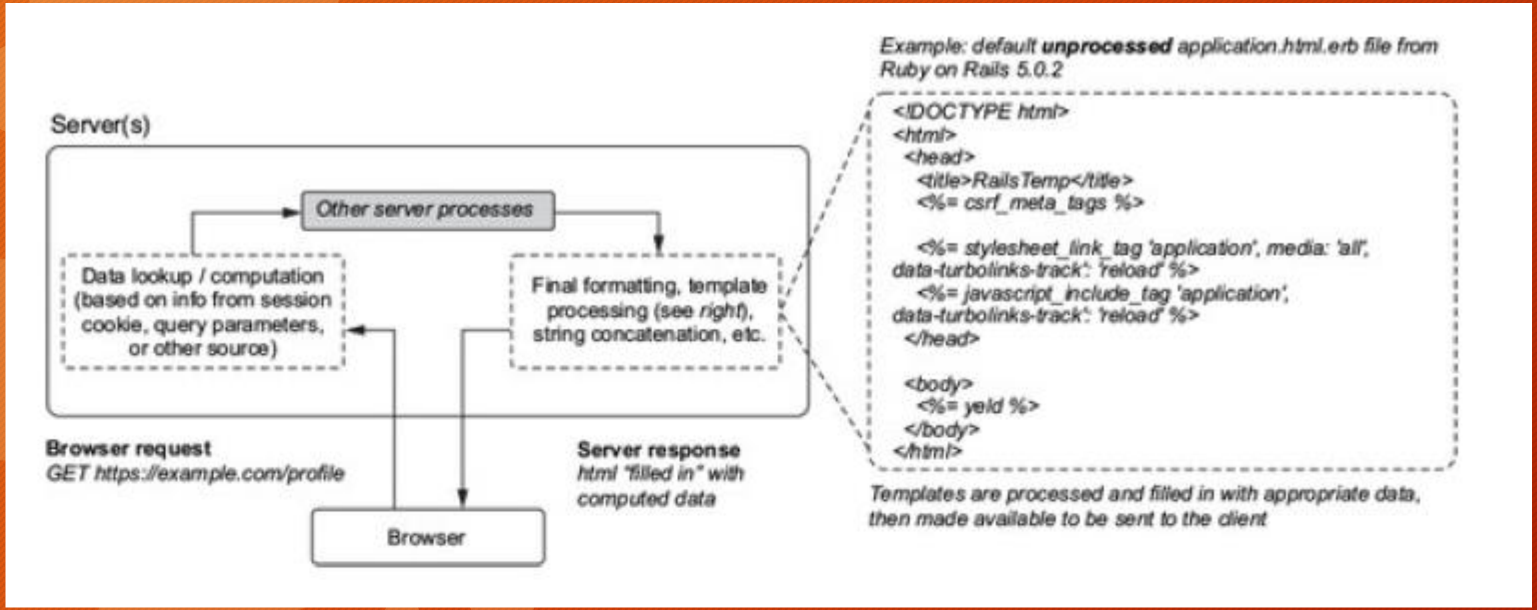

Scheduled task
Step 3. Send the push notification when required
Page
Service Worker
Server
Step 1. Send the subscription details to the server
Step 2. Save the subscription details

Client Side Rendering

• Client -


Server(s)
- Serve HTML + CSS + JS (*if not over cdn*)
- RESful API serves JSON
- interact with database, other normal server tasks
Browser request
GET https://example.com/profile
Server response
HTML, additional assets (JS, CSS)
JSON (for API interaction)
Futher requests to API
(non-rendering)
Browser
- Render, manage UI
- Make futher API requests
- (Usually) handle client-side routing

Server Side Rendering


Server(s)
Other server processes
Data lookup / computation (based on info from session cookie, query parameters, or other source)
Final formatting, template processing (see *right*), string concatenation, etc.
Browser request
GET https://example.com/profile
Server response
html "filled in" with computed data
Browser
*Example: default **unprocessed** application.html.erb file from Ruby on Rails 5.0.2*

```
<!DOCTYPE html>
<html>
 <head>
   <title>RailsTemp</title>
   <%= csrf_meta_tags %>

   <%= stylesheet_link_tag 'application', media: 'all',
data-turbolinks-track': 'reload' %>
   <%= javascript_include_tag 'application',
data-turbolinks-track': 'reload' %>
 </head>

 <body>
   <%= yeld %>
 </body>
</html>
```

Templates are processed and filled in with appropriate data, then made available to be sent to the client

Why Render On Server ?

- SEO Support
- Sites that render dynamic content without requiring the DOM tend to fair better
- App needs to show content to users as quickly as possible
- SSR can potentially contribute to a faster speed index by allowing more of your site to be renderable by the browser earlier in the loading process
- Speed Index
  - This metric can be useful for understanding at an aggregate level how quickly a given page appears to load for a user

Server side Rendering - Complexity

- Synchronization
- The client and server operate within different paradigms that don’t always easily map to one another (for example, no DOM, no filesystem, and so on)
- Finetuning SSR can require special tuning of your client and server

isomorphic app architecture

- Simplified and improved SEO—bots and crawlers can read all the data on page
- load.
- Performance gains in user-perceived performance.
- Maintenance gains.
- Improved accessibility because the user can view the app without JavaScript.

Challenges and trade-offs

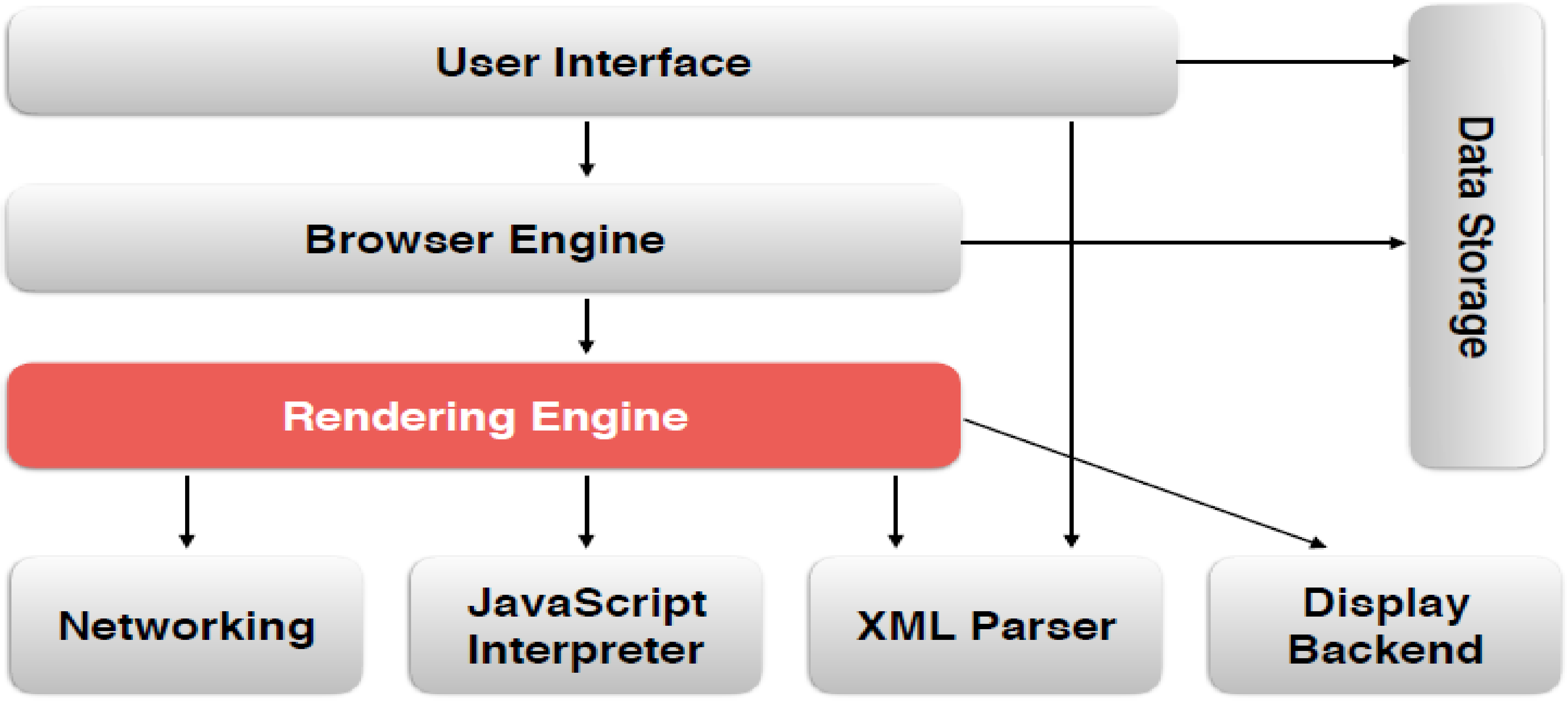
- Handling the differences between Node.js and the browser
- Debugging and testing complexity
- Managing performance on the server

RENDERING COMPONENTS ON THE SERVER

- Api – ReactDOMServer (import ReactDOM from 'reactdom/server')
- renderToString
- renderToStaticMarkup
- renderToNodeStream
- renderToStaticNodeStream

How Browsers Work

Main Browser Components



Source: "A Reference Architecture for Web Browsers" by Alan Grosskurth and Michael Godfrey

Some TCP knowledge

- TCP is designed to probe the network to figure out the available capacity
- TCP does not use the full capacity from the start
- TCP Slow Start - for congestion control and avoidance
- For the first roundtrip ~ 14KB (10 TCP packets)

Rendering Engines

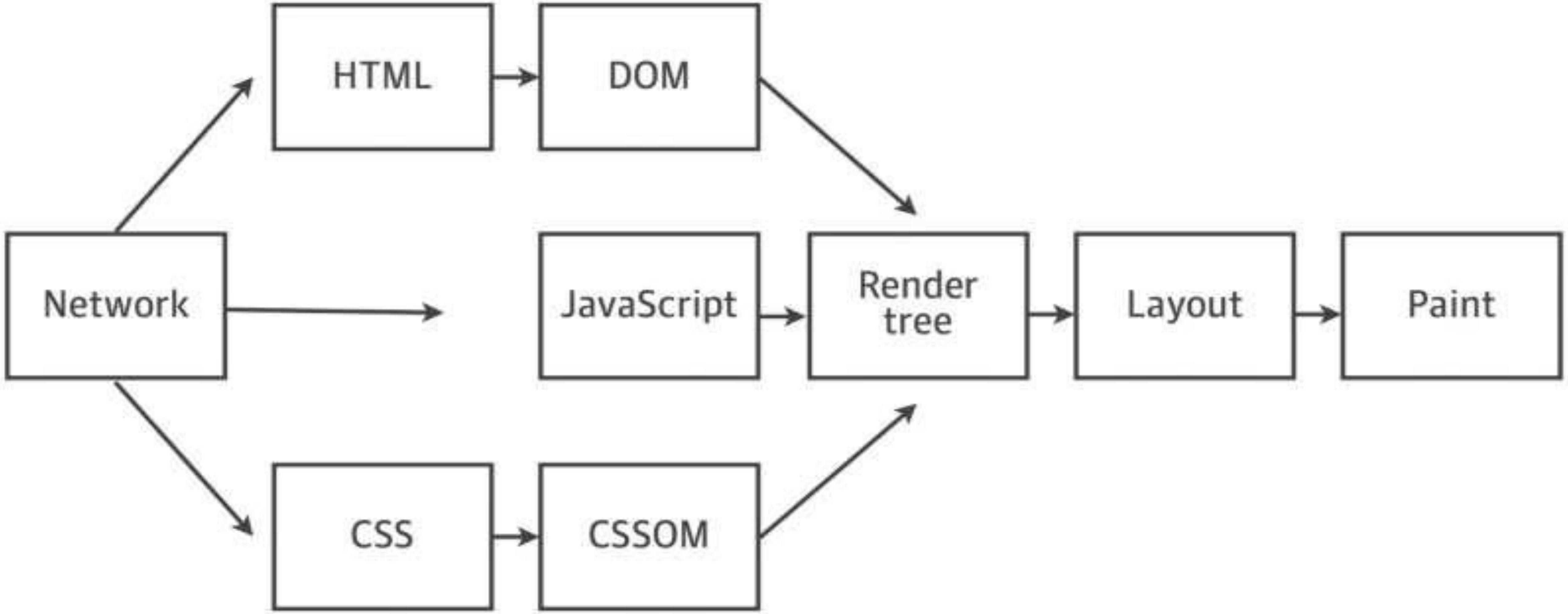
Browser	Rendering Engine
Safari	WebKit
Chrome	Blink
Opera	Blink
FireFox	Gecko
Internet Explorer	Trident
Edge	EdgeHtml

What is Critical Rendering Path

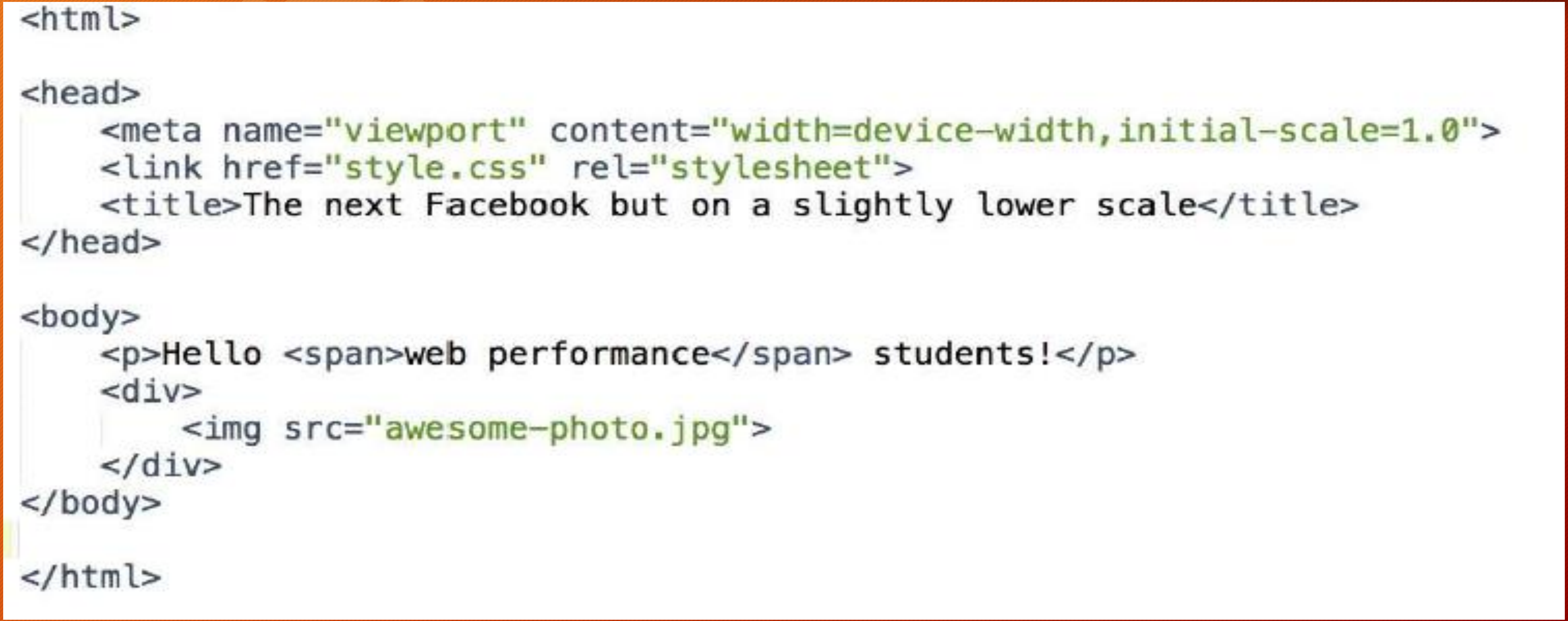
"what happens in these intermediate steps between receiving the HTML, CSS, and JavaScript bytes and the required processing to turn them into rendered pixels - that’s the critical rendering path.”

Critical Render Path

- critical - absolutely needed
- render - display or show (in our case our webpages are "rendered" when they can be seen by a user)
- path - the chain of events that lead to our webpage being displayed in a browser
- initial view - also known as "above the fold", the initial view is the part of a webpage visible to a user before they scrol

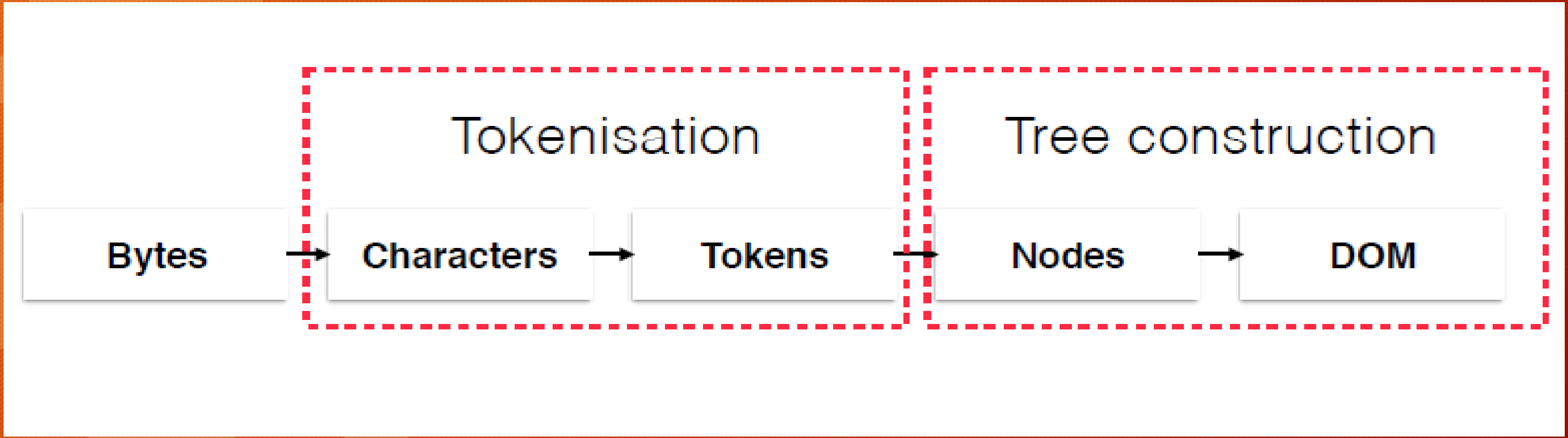

HTML
DOM
Network
JavaScript
Render tree
Layout
Paint
CSS
CSSOM

let’s get the index.html

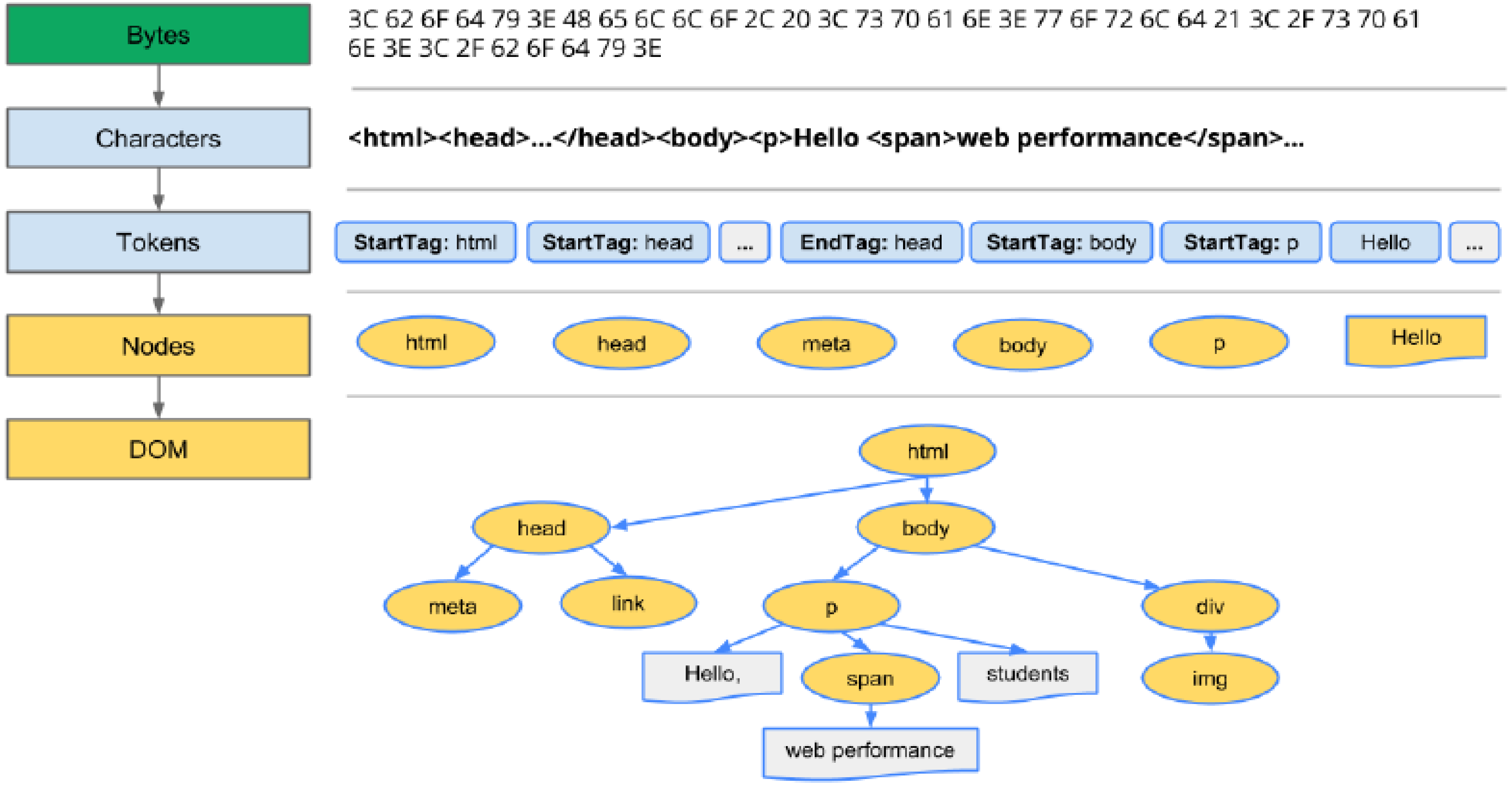


```html
<html>

<head>
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link href="style.css" rel="stylesheet">
    <title>The next Facebook but on a slightly lower scale</title>
</head>

<body>
    <p>Hello <span>web performance</span> students!</p>
    <div>
        <img src="awesome-photo.jpg">
    </div>
</body>

</html>
```

HTML Parser



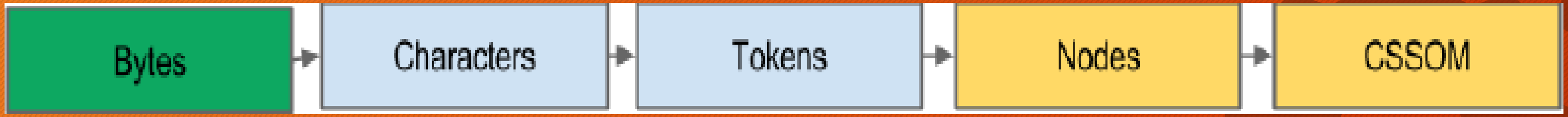
$$\begin{tikzcd}\text{Bytes} \arrow[r] & \text{Characters} \arrow[r] & \text{Tokens} \arrow[r] & \text{Nodes} \arrow[r] & \text{DOM}\end{tikzcd}$$


Bytes
3C 62 6F 64 79 3E 48 65 6C 6C 6F 2C 20 3C 73 70 61 6E 3E 77 6F 72 6C 64 21 3C 2F 73 70 61 6E 3E 3C 2F 62 6F 64 79 3E
Characters
<html><head>...</head><body><p>Hello <span>web performance</span>...
Tokens
StartTag: html
StartTag: head
...
EndTag: head
StartTag: body
StartTag: p
Hello
...
Nodes
html
head
meta
body
p
Hello
DOM
html
head
body
meta
link
p
div
Hello,
span
students
img
web performance

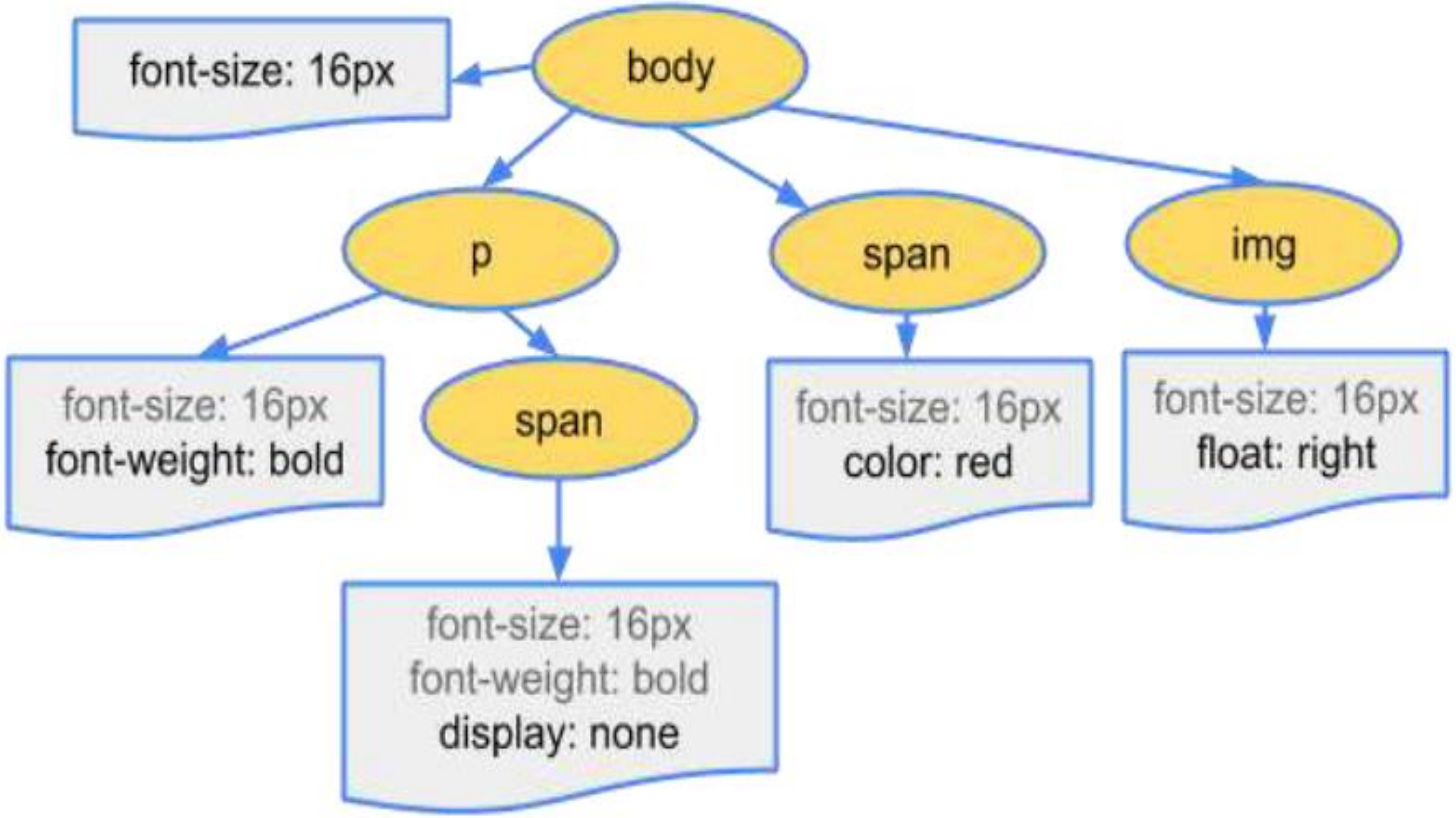
What Browser’s Do

- Conversion: from raw bytes to individual characters (based on encoding - ex: UTF-8)
- Tokenizing: converts strings of characters into distinct tokens specified by the W3C HTML5 standard
- Lexing: tokens are converted into “objects” which define their properties and rules
- DOM construction: relation between tags, tree-like construction

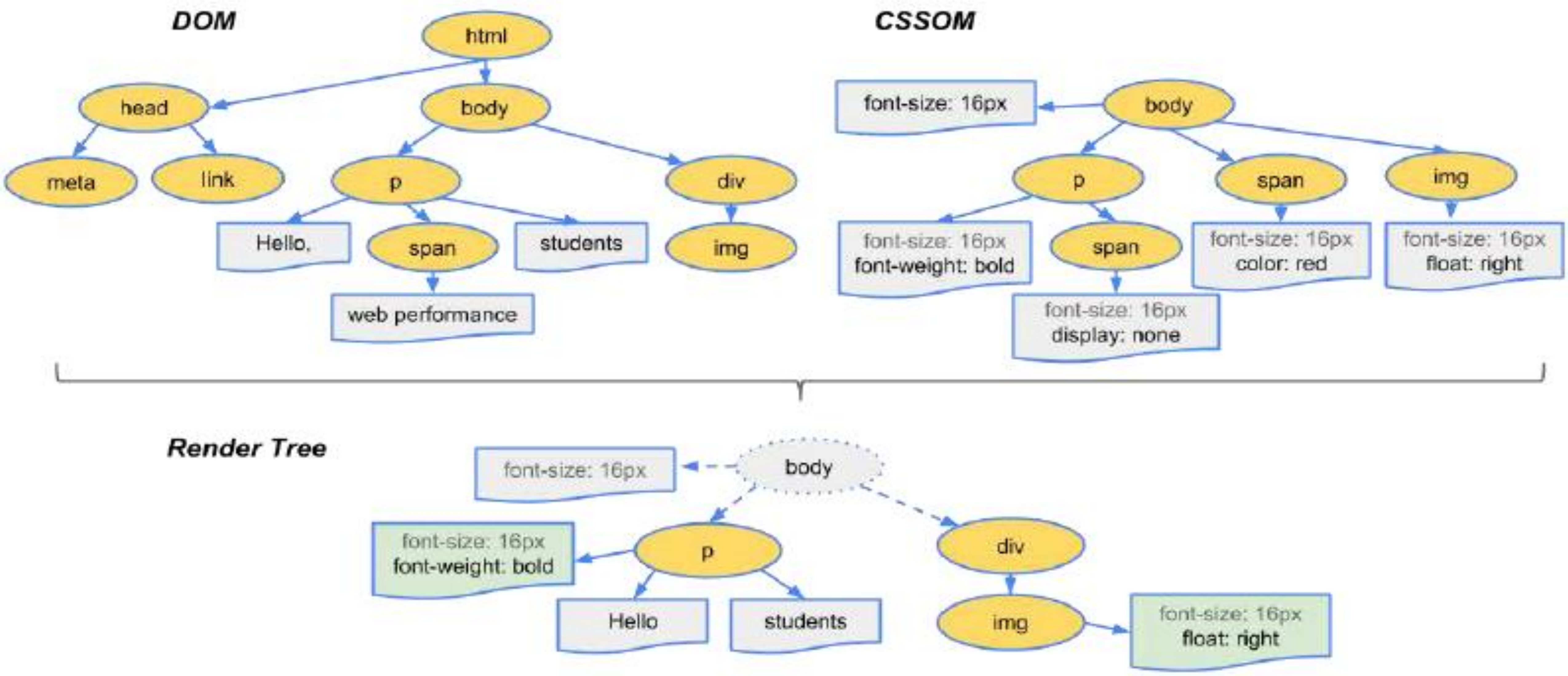
same for CSS …



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



Render Tree


DOM
html
head
body
meta
link
p
div
Hello,
span
students
img
web performance
CSSOM
font-size: 16px
body
p
span
img
font-size: 16px
font-weight: bold
span
font-size: 16px
color: red
font-size: 16px
float: right
font-size: 16px
display: none
Render Tree
font-size: 16px
body
font-size: 16px
font-weight: bold
p
div
Hello
students
img
font-size: 16px
float: right

Render, Layout, Paint

- Final stages before getting something on screen
- The Layout calculates the exact position and size of the elements in the viewport
- The render tree construction, position and size calculation are captured with the “Layout” event in the Timeline (Chrome Dev Tools)
- Paint - converts the render tree to actual pixels

Recap

- Process HTML markup and build the DOM tree
- Process CSS markup and build the CSSOM tree
- Combine the DOM and CSSOM into a render tree
- Run layout on the render tree to compute geometry of each node
- Paint the individual nodes to the screen

Optimizing the critical rendering path is the process of minimizing the total amount of time spent in steps one through five

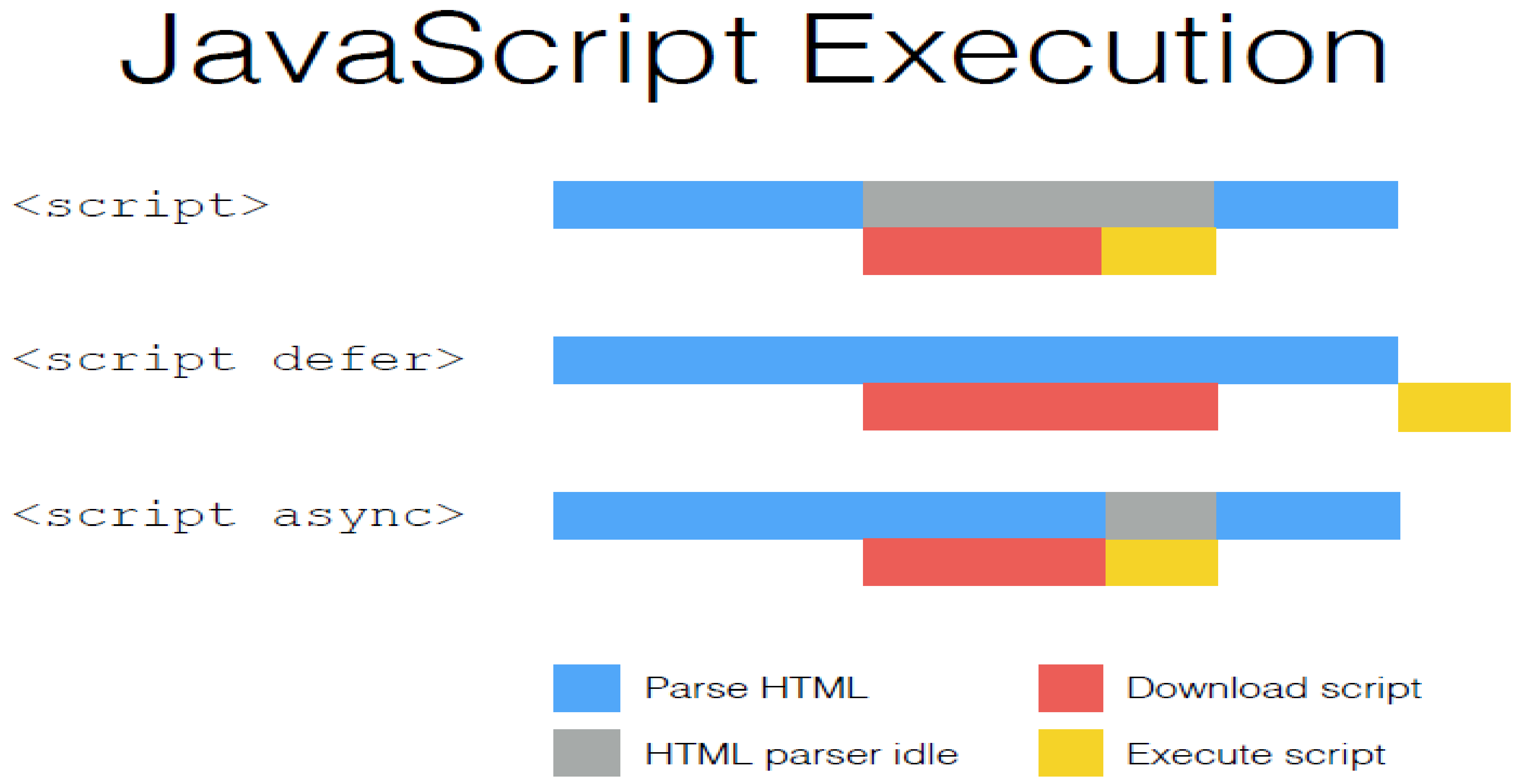
render-blocking - Critical CSS

- Render blocking CSS delays a webpage from being visible in a timely manner.
- Every one of your css files delays your page from rendering.
- The bigger your css, the longer the page takes to load.
- The more css files you have, the longer the page takes to load.

JavaScript Execution

- The location of the script in the document is significant
- DOM construction is paused when a script tag is encountered and until the script has finished executing (JS is parser-blocking)
- JavaScript can query and modify the DOM and CSSOM
- JavaScript execution is delayed until the CSSOM is ready

JavaScript Execution



Perf rules to keep in mind

- HTML is parsed **incrementally**
- CSS is render blocking (can’t construct the CSSOM and the render tree without it)
- CSS is **not incremental** (we must wait for the entire file to download)
- Get CSS down to the client as fast as possible
- Eliminate blocking JavaScript from the CRP (your friend, async)
- Aim for the fastest domContentLoaded time (DOM and CSSOM are ready)

Wrapping Up Optimization Rules

- Minimize use of render blocking resources (specifically CSS)
  - Use media queries on link tags
  - Split stylesheets and inline critical CSS
  - Combine multiple CSS files
- Minimize use of parser blocking resources (JavaScript)
  - Use defer attribute on the script tags
  - Use async attribute on the script tags
  - Inline JavaScript and move script tags to the bottom of the document

Image Optimizations

Understanding Image Types and Applications

- Raster images
  - Sometimes called “Bitmaps”
  - Ex: JPEG,PNG , Gif
  - comprise pixels aligned on a two-dimensional grid.
- Vector Images
  - commonly referred to as SVG
  - These images use a vector artwork format
  - Can be scaled to any size
  - Rasterization :- Every time the image is scaled - their mathematical properties are evaluated, and then they’re mapped to the pixel-based display

Vector vs. Raster images

Vector

- Vector graphics use lines, points, and polygons to represent an image
- Vector images are ideal for images that consist of geometric shapes
- Vector images are zoom and resolution-independent
- Ideal format for high-resolution screens and assets that need to be displayed at varying sizes.

Raster

- Raster graphics represent an image by encoding the individual values of each pixel within a rectangular grid.
- Raster images should be used for complex scenes with lots of irregular shapes and details
- Raster images are zoom and resolution-dependent

Compression Algorithm

- lossy
  - Lossy images use compression algorithms that discard data from an uncompressed image source
  - some level of quality loss is present.
- Lossless Images
  - These image types use compression algorithms that don’t discard data from the original image source
  - lossless formats are perfect when image quality is important.
- categories:
  - *8-bit (256-color) images*
  - *Full color images*

Knowing What Image Formats to Use

Image Format	Colors	Image Type	Compression	Best Fit
PNG	Full	Raster	Lossless	Content that may or may not require a full range of colors. Quality loss is unacceptable and/or the content requires full transparency
PNG (8 bit)	256	Raster	Lossless	Content that may not require a full range of colors but might require single-bit of transparency – such as icons and pixel art
GIF	256	Raster	Lossless	Same as 8-bit PNG with somewhat lower-compression performance , but supports animation
JPEG	Full	Raster	Lossy	Content that requires a full range of colors and for which quality loss and lack of transparency and acceptable – such as photographs
SVG	Full	Vector	uncompressed	Content that may or may not require a full range of colors

Google Advice on Image Optimization

- Eliminate unnecessary image resources
  - **CSS effects** (gradients, shadows, etc.) and CSS animations can be used to produce resolution-independent assets that always look sharp at every resolution and zoom level, often at a fraction of the bytes required by an image file
  - **Web fonts** enable use of beautiful typefaces while preserving the ability to select, search, and resize text - a significant improvement in usability.

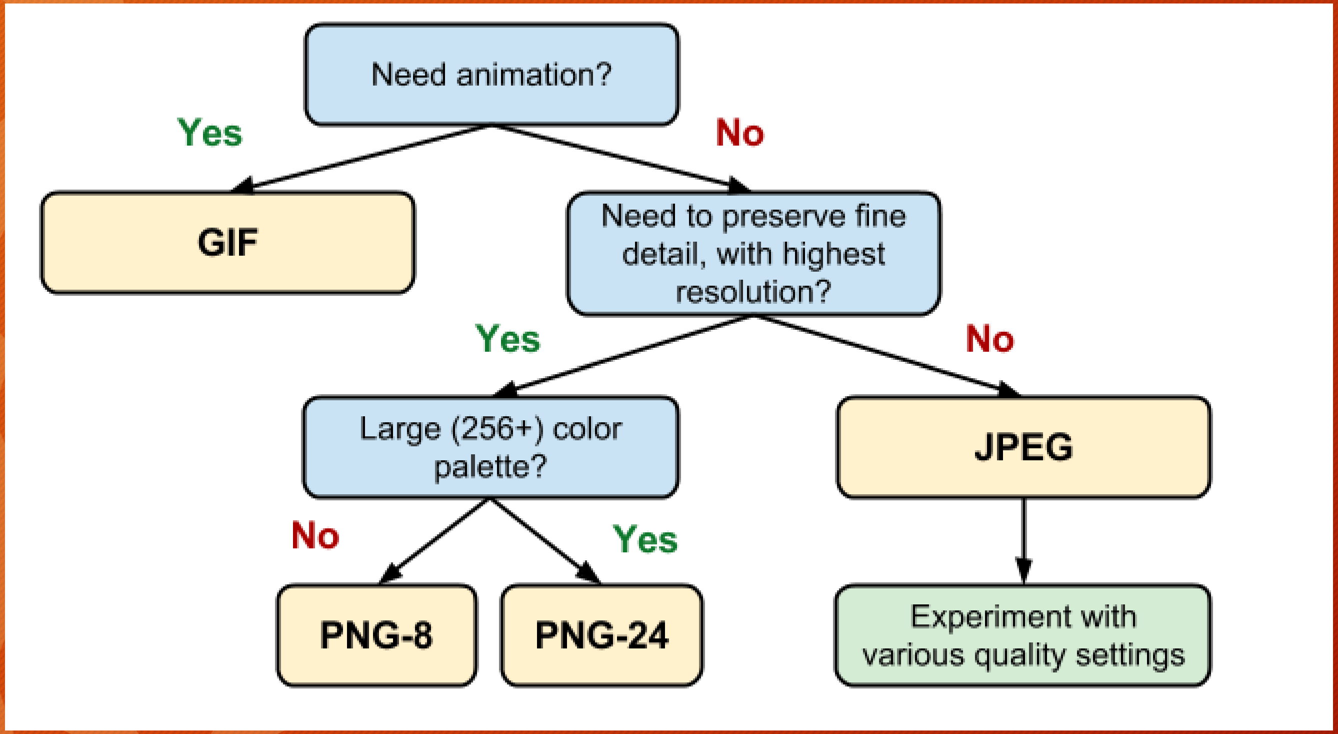
Optimizing vector images

- SVG is an XML-based image format
- SVG files should be minified to reduce their size
- SVG files should be compressed with GZIP

Optimizing Raster Images

- A raster image is a grid of pixels
- Each pixel encodes color and transparency information
- Image compressors use a variety of techniques to reduce the number of required bits per pixel to reduce file size of the image

Selecting the right image format


Need animation?
Yes
No
GIF
Need to preserve fine detail, with highest resolution?
Yes
No
Large (256+) color palette?
JPEG
No
Yes
PNG-8
PNG-24
Experiment with various quality settings

Delivering scaled image assets

- Image optimization criterias
  - optimizing the number of bytes used to encode each image pixel
  - optimizing the total number of pixels
- Ensure that we are not shipping any more pixels than needed to display the asset at its intended size in the browser
  - Otherwise Browser to rescale larger assets
    - consumes extra CPU resources - and display them at a lower resolution
- **One should ensure that the number of unnecessary pixels is minimal, and that your large assets in particular are delivered as close as possible to their display size**

Image optimization: Choose the right format, compress carefully and prioritize critical images over those that can be lazy-loaded

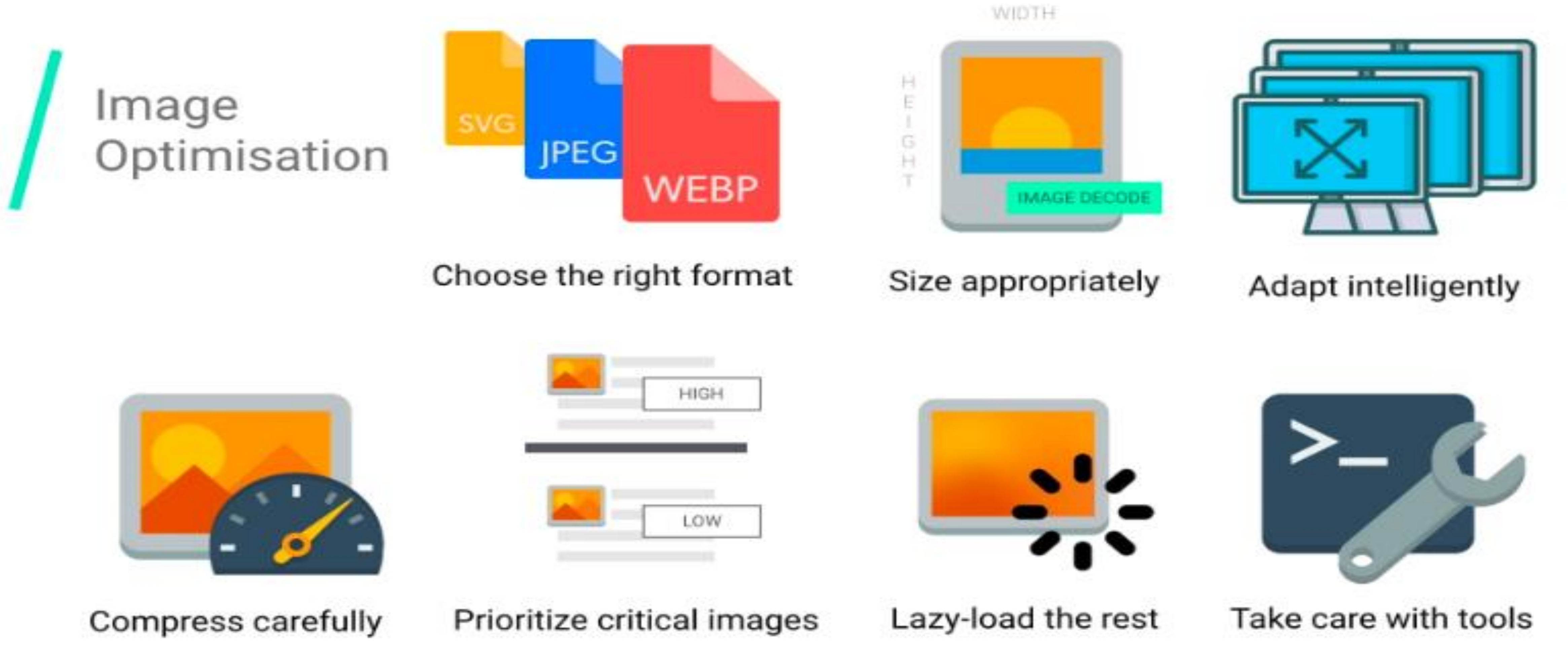


Image optimization checklist

- **Prefer vector formats: vector images are resolution and scale independent, which makes them a perfect fit for the multi-device and high-resolution world.**
- **Minify and compress SVG assets:** XML markup produced by most drawing applications often contains unnecessary metadata which can be removed; ensure that your servers are configured to apply GZIP compression for SVG assets.

- **Minify and compress SVG assets:** XML markup produced by most drawing applications often contains unnecessary metadata which can be removed; ensure that your servers are configured to apply GZIP compression for SVG assets.

- **Serve scaled images:** resize images on the server and ensure that the "display" size is as close as possible to the "natural" size of the image. Pay close attention to large images in particular, as they account for largest overhead when resized
- **Automate, automate, automate:** invest into automated tools and infrastructure that will ensure that all of your image assets are always optimized.
- **Experiment with optimal quality settings for raster formats:** don't be afraid to dial down the "quality" settings, the results are often very good and byte savings are significant

What Is Lazy Loading?

- loading images on websites asynchronously
  - If users don’t scroll all the way down, images placed at the bottom of the page won’t even be loaded
- Lazy Loading Using the Intersection Observer API
  - MDN - *The Intersection Observer API provides a way to asynchronously observe changes in the intersection of a target element with an ancestor element or with a top-level document's viewport*
  - what is being asynchronously watched is the intersection of one element with another
  - IntersectionObserver is an async non-blocking API!
  - IntersectionObserver replaces our expensive listeners on scroll or resize events.
  - IntersectionObserver does all the expensive calculations like getClientBoundingRect() for you so that you don’t need to.

Lazy Loading

- **Robin Osborne's Progressively Enhanced Lazy Loading**
  - The progressive enhancement approach ensures users always have access to content.
  - Not only does it cater for a situation where JavaScript is not available, but also for those cases when JavaScript is *broken*: we all know how error-prone scripts can be, especially in an environment where a significant number of scripts are running.
  - It lazy loads images on scroll, so not all images will be loaded if users don't scroll to their location in the browser.
  - It doesn't rely on any external dependencies, hence no frameworks or plugins are necessary.
- **Lozad.js**
  - Lozad is a highly performant, light and configurable lazy loader in pure JavaScript with no dependencies
  - lazy load images, videos, iframes, and more and it uses the Intersection Observer API

Lazy Loading with Blurred Image Effect

- Lazy Loading with Blurred Image Effect
  - The first thing you see is a blurred, low-resolution copy of the image, while its high-res version is being lazy loaded:
- YALL.js
  - Yall is a feature-packed lazy loading script for images, videos, and iframes
  - Yall uses the Intersection Observer API and smartly falls back on traditional event handler techniques where necessary
  - Great performance with the Intersection Observer API
  - Fantastic browser support (it goes back to IE11)
  - No other dependencies necessary.