Guide to optimize Web app Performance

1. Code Splitting and Lazy Loading

Code splitting :

Code splitting breaks down large files into smaller, manageable chunks loaded on demand. This approach improves initial load time and reduces the total download size, particularly helpful for single-page applications (SPAs) with heavy JavaScript code.

- **Implementing code splitting :**

  Webpack is a popular tool that enables code splitting. You can specify dynamic imports that are only loaded when needed, minimizing the initial load time.

  import(/* webpackChunkName: "myChunk" */ './myModule').then(module => {module.default();});

- React example : React's `React.lazy()` and `Suspense` work together to load components as they're needed.

  import React, { lazy, Suspense } from 'react';

  const LazyComponent = lazy(() => import('./LazyComponent'));

  function App() {

  return (

  <Suspense fallback={<div>Loading...</div>}>

  <LazyComponent />

  </Suspense>);}

Lady Loading :

Lazy loading delays the loading of non-essential resources until they are needed, helping to save data and improve performance, particularly on mobile devices.

- **Image Lazy Loading** : Use the `loading="lazy"` attribute on images to defer loading until they appear in the viewport.

  <img src="placeholder.jpg" data-src="actual-image.jpg" loading="lazy" alt="description">

- **Component Lazy Loading :** Defer loading non-critical components to speed up the initial render. This can be applied to anything from non-visible modals to low-priority pages.

TIP: Prioritize above-the-fold content in the initial load, deferring below-the-fold content using lazy loading to improve the First Contentful Paint (FCP) and Largest Contentful Paint (LCP) metrics.

2.Implementing Caching Mechanisms

Browser Caching :

Browser caching allows static assets like CSS, JavaScript, and images to be stored on the client side for a set duration, reducing the number of server requests.

- **Setting Cache-Control Headers :** Configure headers to specify how long assets should be cached on the client side.

  app.use(express.static('public', { maxAge: '1d', etag: false, }));

TIP: Use strong caching for infrequently changing resources and short-term caching for dynamic resources to ensure users receive up-to-date content.

Service Workers and Offline Caching :

Service workers cache data for offline use and can intercept network requests to serve cached resources when the user is offline

- **Example:** A service worker caching assets for offline use

```
const CACHE_NAME = 'offline-cache';
const assetsToCache = ['/index.html', '/styles.css', '/script.js'];

self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then((cache) => cache.addAll(assetsToCache))
  );
});
```

TIP: For progressive web apps (PWAs), use service workers to cache essential resources for a smooth offline experience.

APIs and Data Caching :

Cache responses from APIs (e.g., REST, GraphQL) on the client-side to reduce redundant network calls.

- Example with IndexedDB and localStorage: Use IndexedDB or localStorage to store API responses.

```
async function fetchData(apiUrl) {
  const cachedData = localStorage.getItem(apiUrl);
  if (cachedData) {
    return JSON.parse(cachedData);
  }
  const response = await fetch(apiUrl);
  const data = await response.json();
  localStorage.setItem(apiUrl, JSON.stringify(data));
  return data;
}
```

3.Using Performance Auditing Tools :

Lighthouse Audits

Lighthouse is a powerful tool integrated into Chrome DevTools, which provides insights into performance, SEO, accessibility, best practices, and PWA optimization.

- STEPS :
    1. Open DevTools and go to the "Lighthouse" tab.
    2. Select the categories to audit.
    3. Click "Generate Report" to see scores and recommendations.
- Metrics to Improve
    - ☐ **First Contentful Paint (FCP)**: Reduce the time it takes for the first visual element to load by optimizing critical resources.
    - ☐ **Largest Contentful Paint (LCP)**:Defer non-critical JavaScript, compress images, and use lazy loading.
    - ☐ **Time to Interactive (TTI)**:Optimize JavaScript execution by minimizing blocking tasks.

WebPageTest and PageSpeed Insights

Tools like WebPageTest and PageSpeed Insights offer detailed performance breakdowns, including metrics on Time to First Byte (TTFB), LCP, and JavaScript execution times. Use these tools for deeper insights and additional optimizations.

4.Additional Tips and Tricks for Web App Optimization :

Minimize Http Request

Each asset requested (CSS, JS, images) generates an HTTP request, so minimizing these improves load time.

- **Combine CSS/JS Files**: Use Webpack or other bundlers to combine smaller CSS or JS files into fewer files, reducing HTTP requests.
- **Remove Unused CSS/JS**: Use tools like PurgeCSS to remove unused CSS, reducing file sizes.

Use CDN (Content Delivery Network)

Serving static assets (like images, CSS, and JS files) from a CDN reduces the load on your server and improves load time for users by serving assets from a server closer to their location

Image Optimization

Images are one of the heaviest assets; optimizing them can drastically improve performance.

- **Compression**: Use tools like ImageOptim or TinyPNG to reduce image file sizes.
- **Responsive Images**: Use `srcset` and `sizes` attributes for responsive images to serve different images based on screen size.

  <img src="image-small.jpg" srcset="image-large.jpg 1024w, image-medium.jpg 768w" alt="example">

- **Next-Gen Formats**:Consider using next-gen formats like WebP, which have superior compression rates

CSS and JavaScript Minification

Minifying your CSS and JavaScript reduces file sizes by removing unnecessary whitespace and comments.

- Use tools like Terser for JavaScript and CSSNano for CSS to automate this process.

Avoid Render-Blocking Resources

Render-blocking CSS and JavaScript delay page rendering, so defer or asynchronously load non-critical scripts.

- **Defer Non-Essential JS**:Use the `defer` attribute on script tags for non-critical JS

  <script src="non-critical.js" defer></script>

SOME KEY AREAS THAT REMAIN UNCOVERED :

1.Leverage Caching for Optimal Data Management

Effective caching can significantly reduce network requests, improve load times, and reduce server load. Here's how to leverage caching more effectively:

Different Caching Layers :

1. **Client-Side Caching**: Cache assets like CSS, JS, and images directly in the user's browser using `Cache-Control` and `Expires` headers. This prevents the need to reload assets each time.
2. **Server-Side Caching**: Cache data at the server level (e.g., using Redis or Memcached). This is particularly effective for dynamic, frequently accessed content, reducing the need for repeated database calls.
3. **Edge Caching with CDNs**: Use Content Delivery Networks (CDNs) to cache and serve static content closer to the user's location, reducing latency and improving speed for global users.

Tip : For APIs, use stale-while-revalidate headers to serve stale content from the cache while fetching fresh data in the background. This approach provides quick responses without compromising freshness.

2.Optimize Images for Faster Load Times

Images often make up the bulk of a page's size, so optimizing them is essential.

- **Image Compression**: Tools like TinyPNG, ImageOptim, or an automated build process with `imagemin` reduce image file size without compromising quality. This lowers bandwidth and load time.
- **Responsive Images with `srcset` and `sizes`**: Use `srcset` and `sizes` attributes in your image tags to serve different image resolutions based on the user's screen size and resolution.

  <img src="image-small.jpg" srcset="image-large.jpg 1024w, image-medium.jpg 768w" sizes="(max-width: 600px) 100vw, 50vw" alt="example">

- **Use Next-Gen Formats**: Formats like WebP and AVIF provide better compression and quality than traditional formats (JPEG, PNG). Serving images in next-gen formats can lead to substantial savings in file size.

- **Lazy Loading Images**: Lazy load images below the fold to save bandwidth and speed up initial load times. The `loading="lazy"` attribute allows browsers to load images as they enter the viewport.

  Tip: Optimize hero images and above-the-fold content with critical image optimization (such as inline base64 encoding for very small images) to ensure faster LCP (Largest Contentful Paint) scores.

3.Minimize Network Request :

Each HTTP request increases load time, so reducing the number of requests is critical. Consider the following methods:

- **Combine CSS and JavaScript Files**: Bundle CSS and JavaScript files to reduce the number of requests. Use tools like Webpack, Rollup, or Parcel to combine and minify these files.
- **Inline Critical CSS**: Instead of loading all CSS at once, inline the essential styles for above-the-fold content directly in the `<head>` to avoid render-blocking. Load non-critical CSS asynchronously.
- **Remove Unused Resources**: Use tools like PurgeCSS to analyze and remove unused CSS from your project, which will reduce the file size and number of assets required.

  Tip: Use prefetch and preconnect links to optimize resources that will be needed soon but aren't immediately critical. Prefetch loads resources during idle time, preparing the page for smoother interactions.

4.Optimize JavaScript Execution and CSS Performance :

JavaScript and CSS are integral to rendering a web app, but large or unoptimized files can be costly in terms of performance.

1. JavaScript Optimization:

- **Defer Non-Critical Scripts**: Use the `defer` attribute on script tags to delay loading of non-essential scripts, allowing content to load first.
- **Reduce JavaScript Payload**: Trim the size of JavaScript files by removing dead code and unused libraries. Tree-shaking with bundlers like Webpack can automatically remove unused code.
- **Use Web Workers**: Offload complex computations from the main thread to background threads using Web Workers, which can significantly improve responsiveness.

2.CSS Optimisation:

- **Minify CSS**: Minify CSS files to reduce size by removing whitespace and comments. Use tools like CSSNano for this purpose.
- **Reduce CSS Complexity**: Avoid overly complex or deep CSS selectors as they slow down rendering. Use flat, straightforward selectors whenever possible.

- **Avoid Layout Thrashing**: Optimize CSS and JavaScript to prevent excessive reflows and repaints (layout thrashing). Repeated changes to the DOM can lead to significant performance hits, so use efficient DOM updates and CSS animations for smooth performance.

  TIP : Use requestAnimationFrame for animations and DOM updates that need to run efficiently with the browser's rendering pipeline. This ensures smoother animations and DOM changes.

OVERVIEW :

1. **Enable code splitting and lazy loading**.
2. **Optimize caching** using browser, API, and service worker caches.
3. **Use Lighthouse** and other audit tools regularly to identify performance bottlenecks.
4. **Minimize HTTP requests** and **combine or remove unused CSS/JS**.
5. **Serve assets via CDN** for faster load times.
6. **Optimize images** by using compression, responsive images, and next-gen formats.
7. **Minify and defer non-essential CSS/JS** to reduce blocking.