# 44 quick tips to fine-tune Angular performance

Angular is a great framework and is well suited for developing large app built to get the highest performance possible today on the web. But sometimes we as a developer ends up doing things which result in the poorly performing app.

In this post, I have put together Angular specific best practices to have best load time and runtime performance. Also, apart from the Angular, I have also mentioned some general best practices in web development. My intention in this post is just to list down the best practices and avoiding the detailed explanation thus keeping the post small.

. . .

## Load Time Performance

**1. AOT:** As opposed to JIT Compilation where the compilation is done in the browser, AOT compiles much of the code during the build process (also called offline compilation) thus reducing much of the processing overhead on the client browser. With angular-cli just specify the "aot" flag (if prod flag is present, then aot flag not required) and AOT will be enabled.

**2. Tree-shaking:** This is the process of removing unused code resulting in smaller build size. If you are using angular-cli, Tree-Shaking is enabled by default.

**3. Uglify:** It is the process where the code size is reduced using various code transformations like mangling, removal of white spaces, removal of comments etc. For webpack use uglify plugin and with angular-cli specify the "*prod*" flag to perform the uglification process.

**4. Google Closure Compiler:** This is the compiler used by Google for their products which results in much smaller bundle size compared to Webpack uglify by performing much more aggressive minification. Even though it is not officially supported by the Angular team, you can look this implementation with closure compiler.

**5. Webpack 4:** Using Webpack 4 (and higher) for your angular-cli or custom webpack build results in more smaller build size compared to Webpack 3. Webpack 4 has mode option which lets you specify the optimization type (production or development) without you requiring to write any explicit configuration giving you the best possible results for the target environment. Also, build time with Webpack 4 is much faster (60% to 98%) than the earlier version thereby reducing the development time.

**6. Prod flag:** For the production, build specify the "*prod*" flag in the angular-cli application. It will enable various build optimizations like uglify, AOT, removal of sourcemaps, service workers (if enabled) producing a much smaller build size.

**7. Build-optimizer flag:** If you are using angular-cli make sure you specify "*build-optimizer*" flag for your production build. This will disable the vendor chunk and will result in more smaller code.

**8. Lazy loading:** Lazy loading is the mechanism where instead of loading complete app, we load only the modules which are required at the moment thereby reducing the initial load time. In simple words, it means "*don't load something which you don't need.*"

**9. Server side rendering:** Rendering the first page of your application on the server (using Node.js, .Net, PHP) and serving it as a static page causes near to instant rendering thus greatly improves perceived performance, speed, and overall user experience. You can use Angular Universal to perform server side rendering.

**10. Progressive Web App:** PWA makes your app load much faster, it gives the offline capability to your app and gives near native app experience thus greatly improving overall perceived performance by the user.

**11. Ivy Render Engine:** Angular team recently announced a new render engine named Ivy. It results in much smaller bundle size than the current engine with improved debugging experience. Though it is still not production ready you can still try it in your app. You can look at this ng-conf keynote for more details.

**12. Updating Angular and angular-cli:** Updating your Angular and angular-cli regularly gives you the benefit of many performance optimizations, bug fixes, new features, security etc.

**13. RxJS 6:** RxJS 6 makes the whole library more tree-shakable thereby reducing the final bundle size. However, it has some breaking changes like operators chaining is not possible instead, *pipe()* function (helps in better tree shaking) is introduced to add operators. They have also renamed some operators.

**14. Service worker cache:** If you have configured your app to support Progressive Web App, make sure to specify all the necessary static resources in the PWA config JSON. These static files will be cached in the client's browser making the second time load much faster.

**15. Cache-control header:** cache-control header controls who caches the response under what condition and for how long thus eliminating the need for network round trip for the resources which are cached.

**16. Third party packages:** Review the third party packages you are using and see if better and smaller alternative is available as it may reduce the final size of your build.

**17. Unnecessary use of third-party packages:** If you include a third-party package just to achieve a small functionality which could be easily done natively with JavaScript or Angular then you are adding unnecessary size overhead to your app which could have been easily saved. For example, if you are including Lodash just to do a simple object filtering then it is totally unnecessary as you could do the same thing natively in JavaScript.

**18. defer attribute:** Mentioning defer attribute to your script tag will defer the loading of the scripts (sychronous) until the document is not parsed thus making your site interactive quicker. For angular-cli app currently there is no way to add this automatically during the build, you have to do it manually after the build.

**19. async attribute:** Just like the defer attribute, async delays the loading of scripts until the document is not parsed but without respecting the order of loading of the scripts. The best example to use it with google analytics script which usually independent of any other scripts.

**20. Gzip compression:** Gzip compression can greatly decrease the size of the response body and hence increase the speed of a web app. Make sure you have enabled gzip compression in your backend. For express.js you can add compression middleware.

```
var compression = require('compression')
var express = require('express')
var app = express()
app.use(compression())
```

**22. Preload and Prefetch attributes:** These attributes help to load the static resources as quickly as possible thus improving the time for first meaningful paint. Having faster first meaningful paint greatly affect the user experience. Preload and Prefetch are almost similar with the only difference is that Preload resources have greater priority. So use Preload for the assets which are essential for the initial rendering and use Prefetch for the resources which are required after site load (required in future across the pages). You can read more about these attributes here.

**23. Updating Third Party Packages:** Make sure you are regularly updating your third party packages. Many times newer packages may contain many performance improvements including smaller size and other build time performance optimizations (e.g. RxJS 6). Also by regularly updating the packages, you may get many improvements related to the bug fixes, security vulnerability fixes, fixes related to package compatibility etc.

**24. Compressing images:** It's a good idea to compress the images without losing much of the quality thereby saving the bytes transferred over the network improving the build time. There are many tools available to achieve this. VS Code extension called TinyPNG can be used to compress Jpeg and PNG images without losing much of the quality.

**25. Remove unused fonts:** It's a good idea to remove the unused fonts which may help you save few bytes over the network.

**26. Slow DNS and SSL:** Sometimes your DNS and SSL provider could be the reason for slow load time. So make sure the DNS and SSL are fast and configured correctly.

## Run Time Performance

**27. Change Detection:** By default on each asynchronous event, Angular performs a dirty checking by performing a change detection for the whole component tree. Such dirty checking could be a lot computation heavy for a medium to large apps. You can drastically reduce the change detection by setting "*ChangeDetectionStrategy*" to "*OnPush*". The OnPush strategy promotes the use of immutable data structures.

**28. Detach Change Detector:** We can completely detach the component from change detection thereby giving a developer the control to inform Angular as to when and where to perform the change detection.

```
import {AfterViewInit, ChangeDetectorRef} from '@angular/co
@Component(…)
class AppComponent implements AfterViewInit {

    constructor(private cdr: ChangeDetectorRef) {}
    ngAfterViewInit() {
        // We only want to detach the change detectors afte
        // performed for the first time
        this.cdr.detach();
    }

```

**29. Web Workers:** The JavaScript implementation in all browser is single threaded thus making the whole app to run on a single thread. Such single-threaded execution drastically reduces the frame rate of the complex application as both UI painting and JS execution handled by the same thread. As Angular by default avoids direct DOM manipulation, it is possible to run the entire Angular app in a separate web worker thread thereby keeping the main thread free to just handle the UI rendering. Check this post to see how to run an angular-cli app inside web worker. However, there are many npm packages which try to access DOM directly thereby creating issues while running the whole app inside the worker process. If you want to run only a piece of code under a web worker thread then look at this npm package.

**30. Webassembly:** Webassembly is a low level assembly like a language enabling near-native performance. WebAssembly aims to execute at native speed by taking advantage of common hardware capabilities available on a wide range of platforms. You can take advantage of Webassembly to run some amount of your web app code with Webassembly. Take a look at this demo app created with angular-cli and wasm. However, you should be aware of the fact wasm is still new and using it sometimes may be tricky considering as of now it has only 4 supported data types (2 floating points and 2 integer points). Also as of now, for most of the cases, performance benefit of wasm is not that huge when you are using it to execute a small piece of code compared to inline JS. Properly evaluate which code you want to move to wasm.

**31. trackBy:** By default, *ngFor identifies object uniqueness by reference. If the object reference is broken by updating the content of the object, Angular removes the related DOM node completely and recreate it again even though the actual change required is for only a small part of the DOM node. This issue can be easily solved by using trackBy.

```
@Component({
    selector: 'app',
    template: `<ul>
                    <li *ngFor="let item of items; trackBy:
               </ul>`
})
class AppComponent {
    Items = [
        {
            id: 1,
            name: 'item 1'
        }, {
            id: 2,
            name: 'item 2'
```

**32. Pure Pipes:** In the "*@Pipe*" decorator you can specify "*pure*" flag as true. This flag indicates that the pipe is not dependent on any outside or global state and is side effect free. This enables Angular to cache the outputs for all the input parameters the pipe has been invoked with and thus allows to reuse the values instead of recomputation. This can lead to a massive reduction in the duplicate operations performed in many cases thus hugely improving the performance.

**33. Avoid complex computations in the template:** Avoid doing complex calculation in the HTML template (ex calling some component method inside the template), instead leverage the use of pure pipes thereby taking advantage of Angular caching and hence avoiding duplicate operations or if the use of pipe is not possible, see the opportunity to pre-calculate the values and then directly bind values instead of calling the component method in the template.

**34. enableProdMode:** Invoking *"enableProdMode()"* avoids Angular from performing additional checks for change detection.

```
import {enableProdMode} from '@angular/core';
if (ENV === 'production') {
    enableProdMode();
}
```

**35. AOT Compilation:** AOT not only improves the build time performance but also the runtime performance of the app.

**36. Optimize Events:** Slower DOM events block change detection until the event is not completed. For example, if you have a click event in your template which is handled in the component and the component itself is calling service method to process it, the change detection will not complete until the control is not returned from the service. If your service is taking more time to perform the intended operation, then it will ultimately slow down the change detection. See the opportunity to optimize your logic to improve the duration or if possible try to move your service logic to separate web worker thread or use wasm if needed.

**37. Unsubscribing Observables:** Observables could create the memory leak issue. So it is better to unsubscribe them when they are not needed anymore. However, you don't have to unsubscribe all the observables used. Unsubscribing explicitly is required when a subscription is created inside a component which is destroyed before the observable completes. Check this SO thread for more info.

**38. Observable share() operator:** If you have subscribed the observable at multiple locations/components, then each subscription will try to produce the data even though the data is duplicate. We can avoid the processing of the duplicate data across subscriptions using the "*share()*" operator.

```
import {Injectable} from '@angular/core';
import {HttpClient} from '@angular/common/http';
import {Observable} from 'rxjs';
import {share} from 'rxjs/operators';
@Injectable()
export class AppService {
    data: Observable < any >;
    constructor(private http : HttpClient) {
        this.data = this.http.get<any> ('apiUrl').pipe(shar
    }
```

**39. Progressive Web Apps:** The PWA not just give you a load time optimization but also the runtime optimizations making your app more responsive, interactive, fast, smooth animations, offline support etc.

**40. Updating Third Party Packages:** Again regularly updating your third party packages may also result in better run time performance.

**41. console.log():** Using console.log() statements in your production code could be a bad idea as it will slow down the performance of the app and also logging objects with console.log() creates memory leak issue. When browser's console window is open, the console.log() execution slows down even further by many times thus impacting site's performance significantly. It's better to completely remove the console.log() statements from your production code or at least have an environment specific conditional logging.

**42. Global Variables:** There are many disadvantages of using global variables and one of them is the memory leak. The variables defined in the global scope won't be cleared until the window is reloaded or tab is closed thus resulting in the memory leak if the global variable is not intended to be used throughout the app. If for some reason you want to have global variables, there are better ways to do it in the Angular.

**43. Event listeners:** Adding event listeners to your DOM node could create memory leak issue. If you forget to remove the listener inside the $destroy event of your directive, it will hold a reference to a DOM node even if it is removed from the document. The DOM tree will then become a "Detached DOM tree" and will leak. Modern JS engines are able to figure most of this situations for you and remove the listeners, but more complex tree hierarchies can challenge even the best GC.

**44. Bad Third Party Packages:** If a bad third party package having performance issues (memory leak, costly js execution, security etc) will ultimately affect your app's performance. So it is always advised to properly review any third party package before using.

So these are all the tips you should follow to write a high-performance Angular app. Hopefully, this will help you to fine-tune your Angular app. Also make sure you do proper performance analysis and audit of your app using different tools available like Chrome/Edge/Firefox JavaScript Profiling tool, Heap snapshots comparison, Chrome Lighthouse etc to make a proper judgment of what exactly causing the issue.