Dead Code elimination in Webpack

Problem Statement

We are given a NextJS/React Application and with many different Modules. These applications internally use bundlers like Webpack/Turbopack for bundling of these modules and utilise Minifiers like Uglify, terser or the NextJS swcMinifier for parsing of these bundles and perform code minification, tree shaking like optimizations.

The issue we face is the unexpected behaviour of these minifiers and the bundlers in removing dead code from our bundles. They seem to not perform tree shaking to the extent it is demanded and hence our bundle sizes remain large inspite of large amount of dead code and unused imports.

Sandbox Setup

The testing sandbox used to create sandbox for analysing bundles and working of the NextJS build process can be accessed from https://github.com/RhythmAgg/webpack-nextjs-pages-sandbox

The sandbox linked contains project tested in pages-router. Sandboxes with app-router as well as projects with custom webpack config have also been tested and issues in these environments researched upon. However given the current Sprinklr's projects utilising pages-router, the major attention is on the linked setup.

Webpack and Tree shaking

The tree shaking is used for dead code elimination in bundlers like webpack, turbopack, rollup etc. It marks the unused exports from the code, evaluates the bundles and **removes the dead code which can be safely discarded without affecting the runtime of the application.**

This means that for tree shaking to work

- Use ES2015 module syntax (i.e. `import` and `export`).
- Ensure no compilers transform your ES2015 module syntax into CommonJS modules (this is the default behavior of the popular Babel preset @babel/preset-env - see the documentation for more details).
- Add a `"sideEffects"` property to your project's `package.json` file.
- Use the `production` `mode` configuration option to enable various optimizations including minification and tree shaking (side effects optimization is enabled in development mode using the flag value).

Its necessary that the ESM structure is utilised. Many popular packages like `lodash` cant be tree shaken given they are build in commonJS. For some of these libraries there are alternatives like `lodash-es` that implement them in ESM format.

NextJS we use `swc` for transpilation and minification. It ensures that the bundle can undergo tree shaking and the issues dont arise because of transpilation to commonJS.

In `production` mode, many of the optimizations like usedExports, minification using `swcMinify` are enabled by default and we dont need to set them explicitely specifically in NextJS. But we can ensure this using custom webpack config (like if we want to add CSS loaders)

```
// next.config.mjs
const nextConfig = {
  webpack(config, { isServer }) {
    // modify and return the package
    return config;
  },
};
```

Side Effects

As per the doc:

> A "side effect" is defined as code that performs a special behavior when imported, other than exposing one or more exports. An example of this are polyfills, which affect the global scope and usually do not provide an export.

The bundler can only remove an unused Module completely if the Module has been marked as side effects free.

If the module is not marked side effects free then even is its exports are not used in the code it will be included in the initial bundle and its further dependencies and imports will be analysed. The parser/minifier may then remove the unreferenced code (some of it) but the unnecessary library imports and extra code will remain in the bundle.

This was heavily tested in the sandboxes Eg.

```
// pages.jsx
import { Rectangle, Cube } from "./Components";

export default function Home() {
  return (
    <>
        <Cube />
      <h1>This is home</h1>
    </>
  )
}
```

```
// Components.js
import Test from './test'; // test3 is an unused import

export function Square() {
  return (
    <>
      <Test />
```

```
      <h1>This is a square</h1>
    </>
  )
}

export function Cube() {
    return <h1>This is Cube</h1>
}

export function Rectangle() {
    return <h1>This is Rectangle</h1>
}
```

In this setup if we set `sideEffects: ["pages/Components.js"]` in our package.json then the Components.js import in pages.js will be removed completely from the initial bundle itself. If we don't set this field (**by default its true**) then the components.js will also be included in the intial bundle and its imports `import Test from './test'` will be further evaluated.

If we use <Rectangle /> in pages.jsx and set `sideEffects: ["pages/test.js"]` then on loading the Components.js module, the bundler will mark Cube and Square as unused exports and since Test too is unused (given its use in Square is not utilised) with sideEffects field set it will be safely removed from our bundle.

`sideEffects` **is much more effective** since it allows to skip whole modules/files and the complete subtree.

`usedExports` relies on terser/minifier to detect side effects in statements. It is a difficult task in JavaScript and not as effective as straightforward `sideEffects` flag. It also can't skip subtree/dependencies since the spec says that side effects need to be evaluated. While exporting function works fine, React's Higher Order Components (HOC) are problematic in this regard.

NOTE: **Well, the term tree shaking is a bit misleading: in the initial step, the dead leaves (e.g., unused functions) of the dependency graph are not removed. Instead, the uglify/swcMinify plugin does this in the second step. In the first**

step, a live code inclusion takes place to mark the source code in a way that the uglify plugin is able to "shake the syntax tree."

Why do we need to specify side effects when our minifier automatically removes unused function calls

Specifying sideEffects let us skip the entire dependency subtree of an unused module. The webpack works in 2 stages: Enabling `optimizations.usedExports` lets the webpack mark the unused exports as UNUSED in the initial bundle. However since it imports can be having side effects like asynchronous data fetching, console logs, global variables and function calls, the minifier has to analyse the module's dependencies further and included those side effects.

The side effects tells the webpack that the unused module is free to remove and there is no need for minifier to analyse its subtree. Hence it improves the build time and helps in decreasing the bundle size since now only the required modules will be included in the bundle.

Possible Solution for local Modules

One possible direction now is to write a script to analyze before the build time which modules are free of side effects and hence can be removed in the bundles if not used. Such modules will be collected at build time using AST analysis of the code to search for side effects and the final list of such modules will be specified in the package.json.

NOTE: This solution is only for local modules since library imports need their own package.json to specify such files. Thus its suggested to use tree shakable libraries only. One such implementation can be referenced from our own UI library https://www.npmjs.com/package/@sprinklrjs/spaceweb. The library has all the components pure and all the index files implemented as pure barrel files. It thus adds `sideEffects: false` in its package.json that enables the projects importing it to only use that particular component and not evaluate other imports. Such barrel packages can also be optimized with nextJS's `optimizePackageImports.`

Side effects to currently look for in the modules(only in the module Scope):

- Call expressions like function calls that are not marked with `/*#__PURE__*/`
- Expressions like console.logs()
- Assignment operations to document, window objects
- If, for/while loop further analysis
- Polyfills specially with `typeof` operator
- Export statements which further use Call expressions
- try...catch blocks

I have adopted the `/*#__PURE__*/` convention in reference to webpack's configuration https://webpack.js.org/guides/tree-shaking/#mark-a-function-call-as-side-effect-free. It tells that if we mark a Call expression with this annotation, the minifier can skip this call. Thus in order to find side effects in the module ourselves, the script skips such annotated calls.

The function scope are not evaluated because unused functions will anyways be removed by the minimizer and wont affect the outside scope and if they are used they will anyways be included along with the side effects they induce.

The script is tested on the testing sandbox as well as on Modules like spaceweb by introducing side effects. The issues arising from these tests are mentioned in https://www.notion.so/Side-Effects-Script-416f4678a945420da9c0c6ecb7ee63ba?pvs=4.

Barrel Packages

Barrel packages are packages that expose a single Module that re-exports all the internal components of that package. Thus the project doesnt need to know the path of specific modules/components and can work with that single exposed endpoint. Eg

```
// spaceweb/esm/index.js
export * from './classNames';
export * from './spacewebProvider';
```

```
export * from './types';
import rtlCSSJS from 'rtl-css-js';
export { rtlCSSJS };

// spaceweb/esm/page.js
import {rtlCSSJS} from './index'
```

If we dont specify sideEffects and import something like `rtlCSSJS` from this file. All the imports will be analyzed. But we cant tree shake easily them since

> If the library is marked as `external`, it remains a black box. The bundler can't do optimizations inside that box because the dependency would be required at runtime instead.

> If we choose to bundle the library together with the application code, tree-shaking will work if the import doesn’t have side effects (`sideEffects` in `package.json`). But this cant be specified for all the packages and hence cant be general solution.

A better solution is to use nextJS provided optimizations like `optimizePackageImports` that takes in an Array of barrel packages that we intend to optimize like

```
const nextConfig = {
  experimental: {
    optimizePackageImports: ['./barrel-package']
  }
}
```

As the documentation says:

> Next.js will analyze the entry file of `barrel-package` and figure out if it’s a barrel file. This process is cheaper than tree-shaking,

> since it only scans the entry barrel files in one pass. It also recursively handles nested barrel files and wildcard exports (`export * from`), and bails out of the process when hitting a non-barrel file.

Hence it internally maps the imports from the barrel-index file to the actual module that directly exports it like

```
import {Debounce} from 'Lodash-es'
// makes it
import {Debounce} from 'Lodash-es/Debounce'
```

Some libraries like `Lodash-es`, `react-icons,` `@mui/material` are configured by default and no need to add them to this array.

Issues with optimizePackageImports

The optimization expects the barrel files to be PURE ie only re-export and have no mixed exports, other expressions like

```
export * from './classNames';
export * from './spacewebProvider';
export * from './types';

console.log('Partial Barrel file')
export function DirectExport() {
    rerturn {...}
}
import rtlCSSJS from 'rtl-css-js';
export { rtlCSSJS };
```

Such files are not resolved by this optimization but can be present in our files. Hence we can try to do this optimization by our own script to also work with partial barrel files but with certain conditions.

Later i got to know that some work was done previously in this direction. So i build upon the previous script and solved the issues with that script. The work done in this direction can be referred from https://www.notion.so/Barrel-Optimization-Script-31b07592fede4da69156a24b3030ec7a

Major Issue

Tree-shaking fails for different imports from the same file across multiple routes

The issue can be understood from:
https://github.com/vercel/next.js/issues/34559

It means that if we have our code like:

```
// strings.ts - shared utility file
export const STRING1 = "string1";
export const STRING2 = "string2";

// pages/test.tsx
import { STRING1 } from '../strings';
...

// pages/test2.tsx
import { STRING2 } from '../strings';
...

// -> JS bundle for /test route includes
// both "string1" and "string2" definitions
```

Even in bundle of test.tsx, STRING2 shows up which should not be the case.

The issue exists in webpack since it cant create multiple copies of the same module and tree shake them seperately. As described by one of the contributors:

> It's not possible to duplicate a module and tree shake it in different ways when they can be potentially loaded at the same

> time. A module could potentially include state (even while that is not the case here) which would be illegal to duplicate.

A workaround which they suggest it to split our code into separate modules for each export. This means ideally which should export only 1 thing from a module.

One more issue faced when exporting Objects:

```
// strings.ts - shared utility file
const STRING1 = "string1";
const STRING2 = "string2";

export const obj = {
    STRING1,
    STRING2
}

// pages/test.tsx
import { obj } from '../strings';
// use obj.STRING1
...

// pages/test2.tsx
import { obj } from '../strings';
// again use obj.STRING2

// Both the bundles will now contain the
// entire obj although only STRING1 was used.
```

This tells that unused object properties, unlike unused functions, dont go under tree shaking easily. So its suggested to use individual exports instead of Exporting the object of things we want to export.

So current approach: Take a utils file and write a code mod to split the exported object/individual exports into separate modules along with all the dependent code. Then

Webpack provides optimizations per entrypoint(runtime given each entrypoint has their own runtime using optimization.runtimeChunk). This together with optimization.usedExports can be used to solve this issue however in Next js Their is only a single entry point “main.js” and hence a single runtime only and unlike SPAs like vite, CRA it includes automatic code splitting built into its router (refer https://nextjs.org/docs/app/building-your-application/upgrading/from-vite#network-waterfalls). Thus this solution wont work for next js.

Issues on webpack:

https://github.com/webpack/webpack/issues/7782

> https://github.com/webpack/webpack/issues/16558

> https://github.com/webpack/webpack/issues/4453

> https://github.com/webpack/webpack/issues/13356

#13356 says that having a single runtime for the chunks makes tree shaking shared module separately not possible given webpack has to assume that the chunks in the same runtime can be loaded together.

So we implemented a codemod for module splitting and import resolution. The codemod is able to solve the problem and we merged the eliminating barrel optimisation script as well thus having a single script to deal with all the issues with tree shaking.

The module splitting and import resolution codemods are described here: https://fluoridated-chips-cc4.notion.site/Split-Modules-and-Import-Resolution-2b0a1595425c4dff87388b278b175f7d

Trial: Rollup

I tried using rollup bundler for building applications. Rollup is not supported by NextJS so i couldnt use the plugins provided by rollup too. I setup a small application in ViteJS to analyze the bundling process in Rollup. To simulate next js environment where automatic code splitting happens based on routes, I dynamically imported my routes and used react-router-dom. Thus the route specific code was loaded only when that route was hit, similar to NextJS. I created a shared components file between 2 routes having 3 exports - square, cube and rectangle. Route1 used cube, route2 rectangle. In the final bundle i noticed that square was removed from components build and Route1, Route2 imported the respective functions from Components file, which had both cube and Rectangle. On the client side for each route both the Route1 specific module as well as Components module were loaded.

This tells that rollup doesnt aggresively concatenate shared modules between different chunks unlike webpack and by default maintains the ES syntax in the builds. Also it has lesser build size because of lesser runtime code and doesnt depend need sideEffects field to remove unused modules. It analyzes the imported modules and remove them on its own is they are pure.

Thus rollup internally performs maximum tree shaking unlike webpack where we have to specify

Eg:

```
//components.js
export function Cube() {
}
export function Sqaure() {
}
export function Rectangle() {
}

// App.js
import Cube from './components'

export default function App() {
    return (
        <h1> Hi </h1>
```

```
    )
}
```

The app doesnt use the Cube function so it gets removed from the App chunk. If we add a console statemnt to the components file

The app chunk is like:

```
import { j as jsxRuntimeExports } from "./index-DYYbswD9.js";
import "./Components-NbFaSt62.js";
function App() {
  return /* @__PURE__ */ jsxRuntimeExports.jsx(jsxRuntimeExports
}
export {
  App as default
};
```

Hence it does automatic analysis of the code.

But rollup is not tree shaking the shared module differently and loads it together along with route specific code and then imports the necessary code from it.