Treeshaking POC + bloat reducer script

Manager: Karan Vyas

Mentor: Gaurav Soni

Buddy: Nikunj Shah

Intern: Rhythm Aggarwal

sprinklr

Why are my bundles so bloated ?

To find this out and provide solution is my project

My project aims to research and find solutions to deal with the problem of application bundle bloating. This problem is faced by all the development teams. It leads to poor performance and degrades the user experience increasing the load time.

What causes bundle bloating

- Bundlers like webpack bundle the application modules into bundles to be shipped to the client.
- These bundles have modules concatenated together, code minified and optimised for better performance and faster load time.
- The major optimisation is **Dead Code elimination (DCE)** which is achieved through tree shaking algorithm
- **Tree shaking** is fundamentally removal of unused code from a bundle hence decreasing its size which means lesser code to load on the client.

Solving Tree shaking became the objective

To analyze the working of tree shaking algorithm in webpack in a NextJS application:

- Multiple sandbox setups to simulate the development environment.
- Different project structures like:
  - App and Pages router
  - Server and Client components
  - Custom and default configuration, etc
- NextJS, Webpack documentations, articles and github discussions
- Other bundlers like Rollup, esbuild using Vite
- Bundler Analyser used to review bundle sizes experiment with the build process.

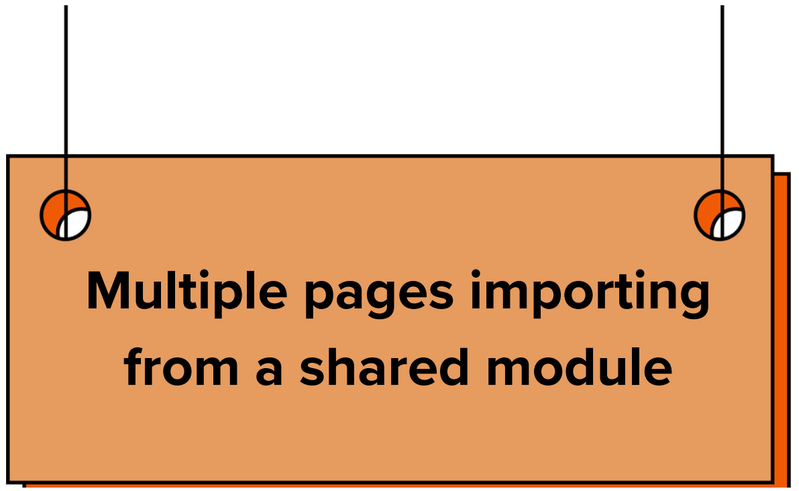
All the research is documented and accessible from: Dead Code Elimination in Webpack

Tree shaking failures in Our Repo

We discovered 2 major cases where tree shaking was failing

CASE - 1

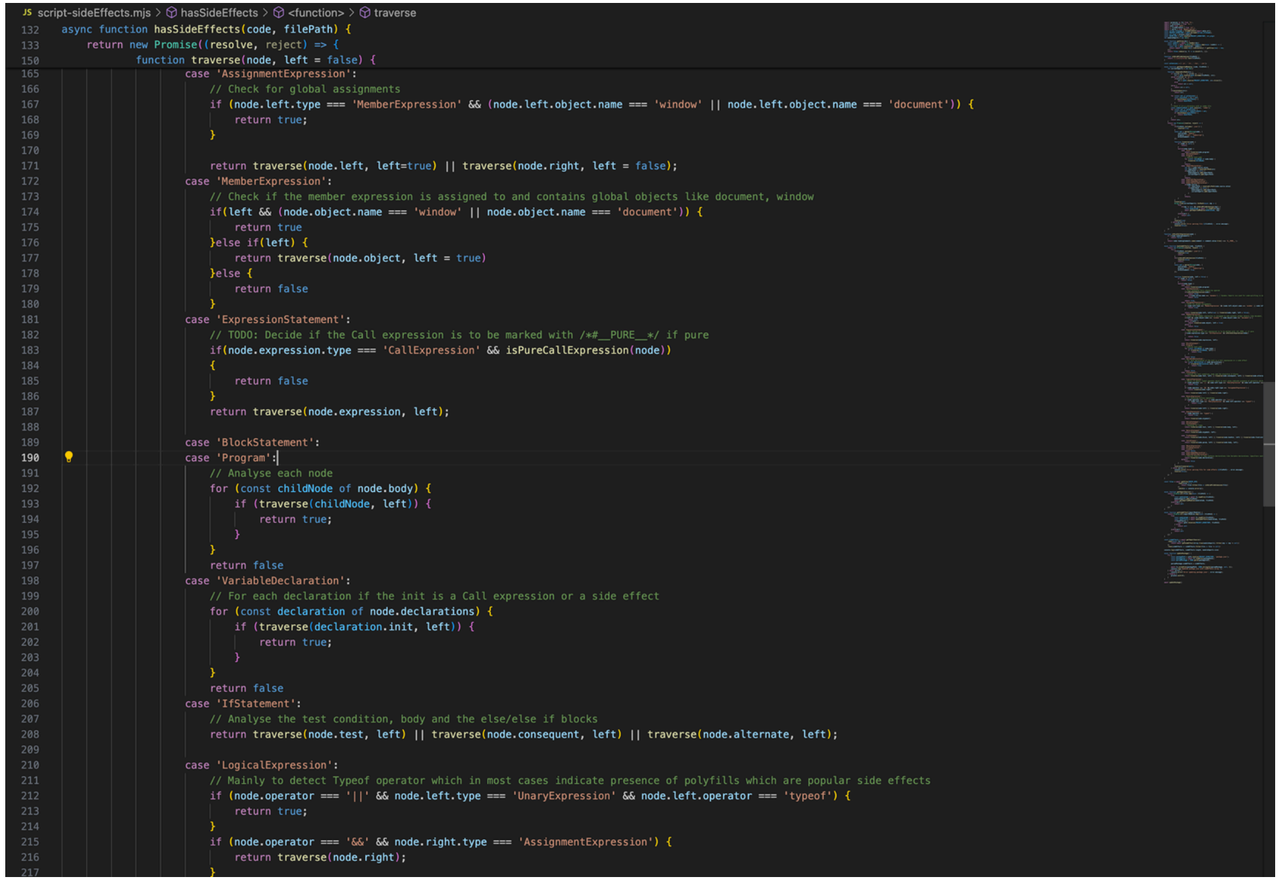
CASE - 2



Initial Approach

- The bundler uses **sideEffects** option in package.json to skip over unused modules and their subtree in a page bundle otherwise they were analyzed further.
    - *sideEffect Modules*: Code that performs a special behavior when imported, other than exposing one or more exports.
- Planned to use this approach for tree shaking barrel files and test for case-2
- Wrote a pre-build node script to parse the **Abstract Syntax Tree** of the package modules and mark them as side-effects free or not.
- Pushed the list of marked modules to package.json

Results - SideEffects Approach



Achievables

- The script identifies many side effects correctly
- The script can be used with pure barrel files

Drawbacks

- It didnt solve case-2 and was not the preferred approach for barrel-exports. Hence the ROI of using it as a prebuild process was very low.
- Function calls like React.forwardRef which are side effects free are also considered as side-effects.

The working, achievables and drawbacks of the script are explained here: Side Effects Script

Case - 1

Imports from a barrel file re-exporting components

```
// 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'
```

A barrel-index file with many re-exports

- We import only one component out of many re-exported by barrel-file
- This causes all the re-export sources to be analysed by the bundler and unwanted code included.
- We looked for directly optimising the imports* to eliminate the need for barrel files.
- An existing script using **jscodeshift** to achieve the same result was analysed but it didn't work and had major drawbacks: Issues with previous codemod
- The new codemod was merged with the solution for case-2 giving scripts solving all the cases simultaneously

*Similar approach adopted by Vercel team optimize Package Imports but it couldnt be used with our package modules.

Case - 2

Multiple pages importing from a shared module

```
// 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
```

- “test.tsx” imports only STRING1 and “test2.tsx” imports only STRING2
- Still the pages contain both STRING1 and STRING2 in their bundle.
- For different routes, the bundler was tree shaking a shared module differently.
- This is the **major issue** causing bundle bloating in the team’s main repo.

Reasons for this behaviour

- The first goal was to study this behaviour and find the reasons for such unexpected issue
- NextJS and webpack github issues, discussions and articles were deeply studied to find a similar issue faced by others and possible solutions
- There were issues on both webpack and vercel forums* where contributors and other explained the issue and its solutions
- Reasons for this to exist are:
  - React applications and frameworks provide single entry points for bundling.
  - Each entry point can have a separate runtime. Hence a **single entry point has a single runtime** for the entire application
  - The bundlers can perform different tree shaking for different runtimes/entry points.
  - For a **single entrypoint** they have to assume that multiple modules may be loaded together hence maintain only a single instance/variant of each module
- This was tested and verified in bundlers like **rollup, esbuild** as well by creating a Vite application sandbox.
- The only viable solution was to **split the shared module into multiple inter-dependent smaller modules exporting a single item only.**

*Github discussion forums mentioning the issue:

- https://github.com/vercel/next.js/issues/34559 - suggests code splitting as the solution
- https://github.com/webpack/webpack/issues/7782 - explains single variant need of a module
- https://github.com/webpack/webpack/issues/13356 - explains single entry/runtime drawback

Proposed solution - Case 1 and 2

Codemods to achieve the following goals:

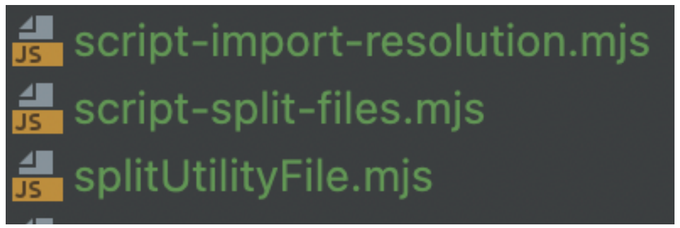
Split the target modules into inter-dependent modules for each exported item.

Create separate modules for shared utilities to avoid code duplicacy and allow variables sharing.

Resolve code importing the target modules to import from the newly created splits

Also achieve the goals of eliminating barrel-exports for solving case 1.

Steps to execute codemods



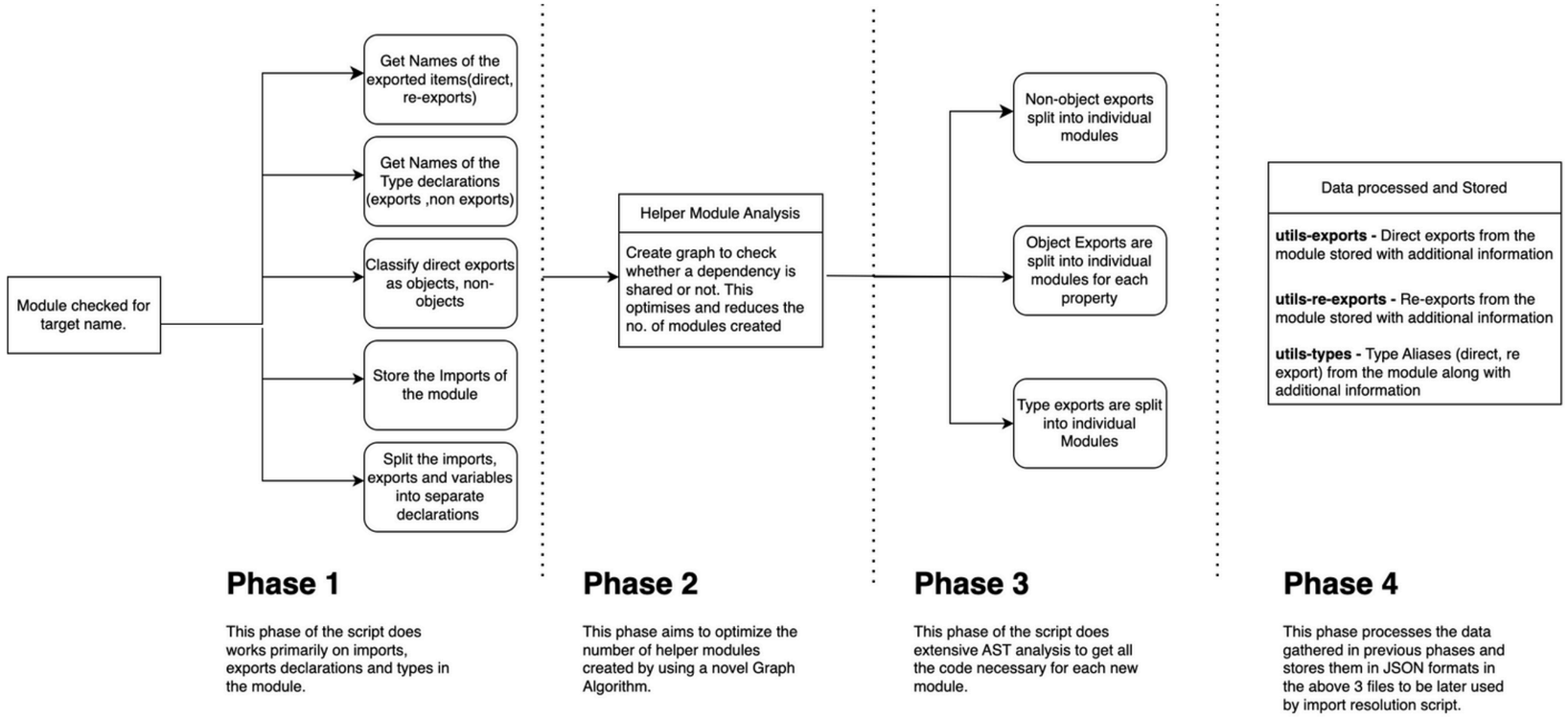
splitUtilityFile script
Wrapper script

1.) **Split files script**
Split Modules codemod

2.) **Import Resolution script**
Import resolution codemod

The wrapper script is executed which internally executes split modules and import resolution codemods sequentially.

Codemod 1: Module Splitting

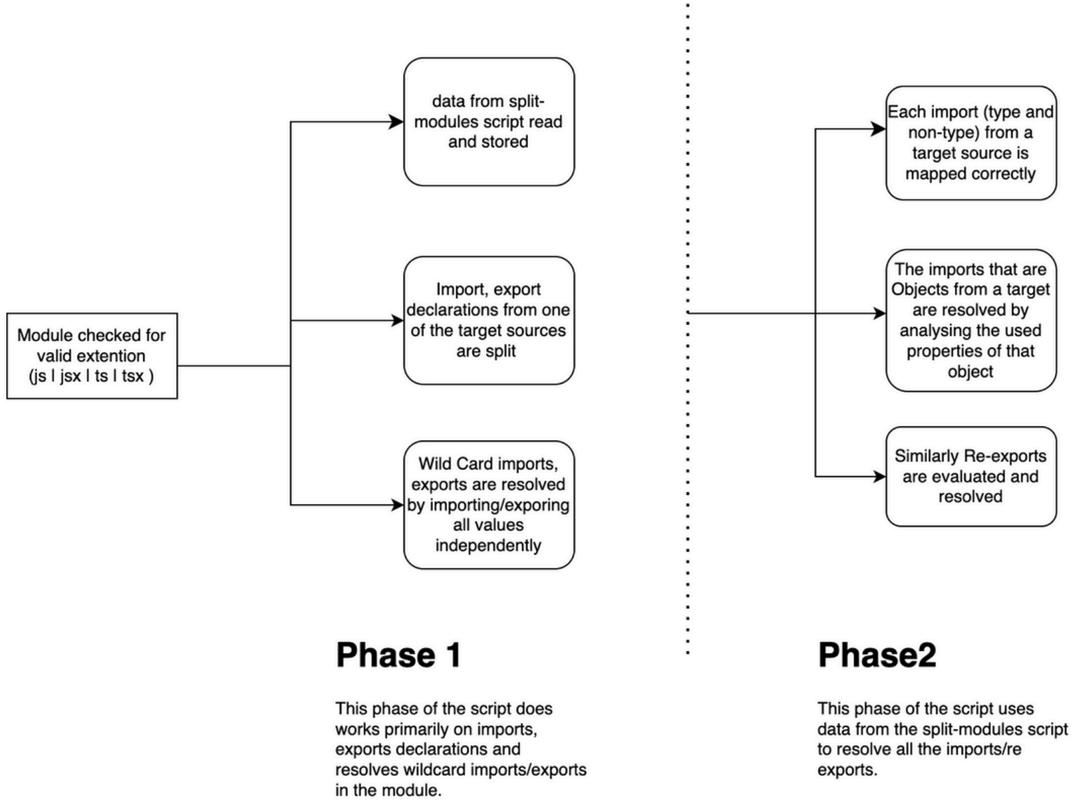


- Script works in **4 phases**
- It achieves:
  - Module splitting
  - Eliminating Barrel-exports
  - Creation of shared variable modules
- It takes target file path and works on modules having both direct and indirect exports.

The script uses many different graph algorithms on AST of the module node for **creating splits** and a noval single-pass algorithm for optimising the **creation of shared modules**.
It writes the processed data to **utils-exports, utils-re-exports, utils-types** JSON files to be used for import resolution further.

Examples of modules covered by the script are discussed here: Module Splitting examples
The code has been documentated for future developers: Documentation - split modules

Codemod 2: Import Resolution



- Script works in **2 phases**
- It achieves:
  - Resolve code importing the target modules to import from the newly created splits
  - Mapping re-exports to the actual soure modules
- It uses the data in JSON files created after codemod 1.
- tsconfig.json is used for resolving aliases
- It eliminates the need of the original target modules

The script runs after the module splitting script on the entire project and uses AST parser to modify the imports of modules to the newly created splits as well as resolving the barrel exports.

Examples of import resolutions by the script are discussed here: Import resolution examples

The code has been documentated for future developers: Documentation - import resolution

Helper scripts

Wrapper Script

- Runs split modules and import resolution script sequentially
- Provides list of command line options
- Used for allowing to run both the codemods with a single command

```
async function runScripts() {
    try {
        console.log(`Running first script: ${firstScript}`);
        const { stderr: firstStderr } = await execPromise(`jscodeshift -t ${firstScript} ${targetModule}`, {maxBuffer});

        if (firstStderr) {
            console.error(`Error in first script: ${firstStderr}`);
            throw new Error('First script failed.');
        }

        console.log('First script executed successfully. Now running the second script.');

        console.log(`Running second script: ${secondScript}`);
        const { stderr: secondStderr } = await execPromise(`jscodeshift -t ${secondScript} ${modifyDir}`, {maxBuffer});

        if (secondStderr) {
            console.error(`Error in second script: ${secondStderr}`);
            throw new Error('Second script failed.');
        }

        console.log('Second script executed successfully.');
        if(mergeRequest) {
          const { stdout: thirdStdout, stderr: thirdStderr } = await execPromise(`node script-gitlab-mr.mjs`);
          if (thirdStderr) {
            console.error(`Error in gitlab-mr script: ${thirdStderr}`);
            throw new Error('gitlab-mr script failed.');
          }else {
            console.log(thirdStdout)
          }
        }
    } catch (error) {
        console.error(`An error occurred: ${error.message}`);
        process.exit(1);
    }
}

try{
    runScripts();
} catch(err) {
    console.error(`An error occurred: ${err.message}`);
    process.exit(1);
}
```

The command details for both scripts are here: Helper scripts

Results - Split Modules and Import Resolution

Achievables

- **Case 1: Imports from a barrel file re-exporting components**
  The script integrate the barrel-elimination logic together with the final codemod. It thus resolves both direct exports and indirect re-exports from a target module.
- **Case 2: Multiple pages importing from a shared module**
  This is the major problem solved by the script. Using code splitting with shared utilities bypasses the need for tree shaking these modules.

The scripts are tested on the main repo. It **saves the manual effort required by the developers to split modules with hundreds of exports and reduces bloated bundles**. Together with the **research documented** and **sandboxes experimented** with during the process, these scripts and codemods can solve the problem of bundle bloating and save future developer's time and efforts to dive deep into the bundling process.

Interesting Challenges faced

Some of the interesting challenges faced during implementing the scripts:

- **Building the dependency graph of each export**
  - Solution: Did scope analysis of the AST nodes to get dependencies and did BFS traversal to recursively get the entire graph
- **Maintain the “declared-first-use-after” principle in the created modules**
  - Solution: Used reverse topological order on the directed dependency graph of the specific export. This ensured declarations(parents) come after dependents(children) in code
- **Ensure that separate modules are created for utilities only when necessary**
  - Solution: Wrote a graph algorithm to mark each utility whether it requires a separate module. If not its concatenated with module using it else imported where needed

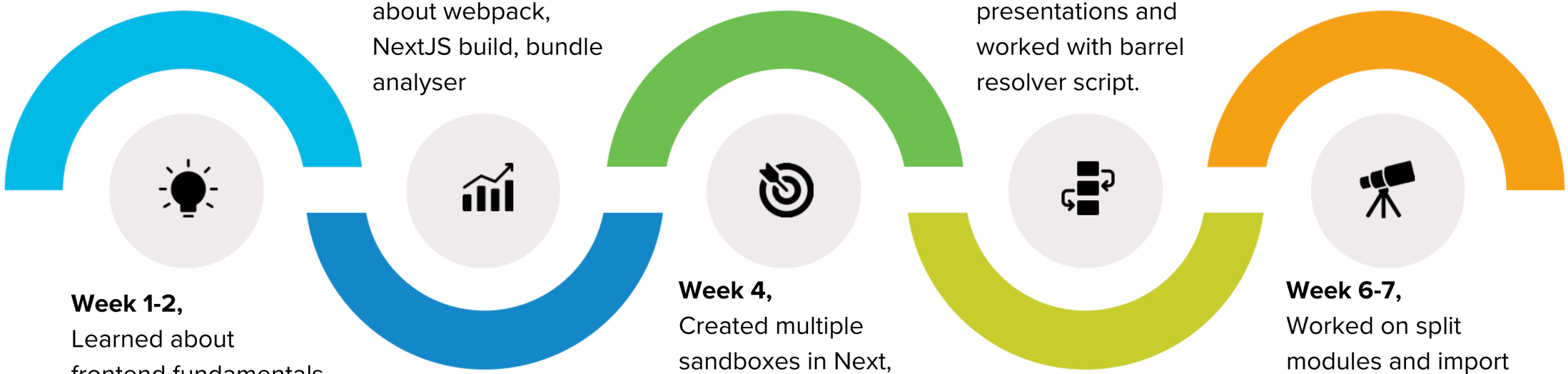
Tests on the main repo

We tested the codemods on both local sandboxes and on the main repository.
We created 2 PRs following the run of the codemods on repository:

- Utility file with object exports: https://prod-gitlab-ui.sprinklr.com/sprinklr/frontend/sprinklr-ui-hub/-/merge_requests/24834
- Utility file with independent exports: https://prod-gitlab-ui.sprinklr.com/sprinklr/frontend/sprinklr-ui-hub/-/merge_requests/24845

These MRs were analysed extensively for errors in the process. We derived that the solution proposed worked in such major cases and provide a stepping stone for all future developments.

My Sprinklr Journey



Week 1-2,
Learned about frontend fundamentals like CSS, JS, React, NextJS, Clean code, GraphQL etc

Week 3,
Started with the project. Learned about webpack, NextJS build, bundle analyser

Week 4,
Created multiple sandboxes in Next, webpack. Tested tree shaking , documented research. Began with side-effects script.

Week 5,
Tested sideEffects script. Had mid-eval presentations and worked with barrel resolver script.

Week 6-7,
Worked on split modules and import resolution codemod. Parallelly tested rollup, esbuild and research github issues

*I created a webapp for accessing my learnings: My Sprinklr Learning


Questions?