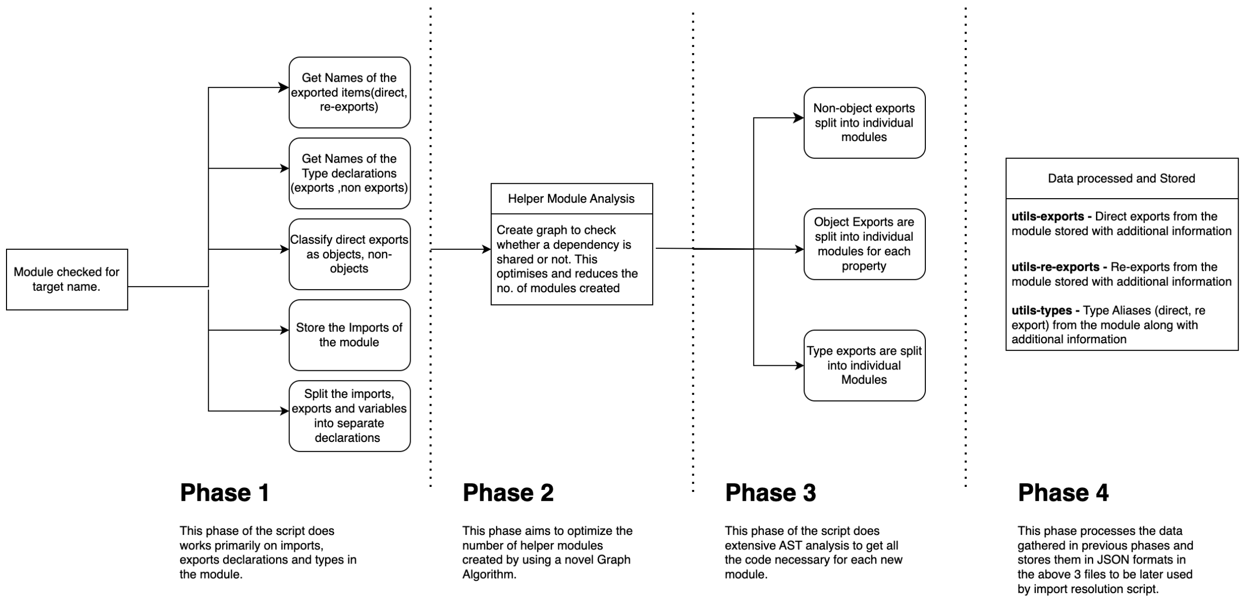
Split Modules and Import Resolution

Split Modules


Module checked for target name.
Get Names of the exported items(direct, re-exports)
Get Names of the Type declarations (exports ,non exports)
Classify direct exports as objects, non-objects
Store the Imports of the module
Split the imports, exports and variables into separate declarations
Helper Module Analysis
Create graph to check whether a dependency is shared or not. This optimises and reduces the no. of modules created
Non-object exports split into individual modules
Object Exports are split into individual modules for each property
Type exports are split into individual Modules
Data processed and Stored
utils-exports - Direct exports from the module stored with additional information
utils-re-exports - Re-exports from the module stored with additional information
utils-types - Type Aliases (direct, re export) from the module along with additional information
Phase 1
This phase of the script does works primarily on imports, exports declarations and types in the module.
Phase 2
This phase aims to optimize the number of helper modules created by using a novel Graph Algorithm.
Phase 3
This phase of the script does extensive AST analysis to get all the code necessary for each new module.
Phase 4
This phase processes the data gathered in previous phases and stores them in JSON formats in the above 3 files to be later used by import resolution script.

Fundamental flow of the script

The script is used to split the utility and barrel files into independent modules and later modify the imports referencing those files in the project. These are the cases handled by the splitting script:

- ▼ A utility file with many independent exports

```
// utils.js
export const x = 'Hi';
export function Hello() {
    ...
}
export type tsType = {
    ...
}

// Modified to
// utils/x.ts
export const x = 'Hi'
// utils/Hello.ts
export function Hello() {
...
}
// utils/tsType.ts
export type tsType = {
    ...
}
```

The new modules are put in a directory created of the same name as the target module. The developer can specify if a separate 'utils' subdirectory is to be created or not. All the modules(exports, helpers, types) are put there.

▼ A utility file with exports that depend on some shared variables

```
// target.js
const function Shared() {
    ...
}

export const x = Shared()

export function Hello() {
    Shared()
}

// Modified to
// target/Shared.js
const function Shared() {
    ...
}
export default Shared

// target/x.js
import Shared from './Shared'
export const x = Shared()

// target/Hello.js
import Shared from './Shared'
export function Hello() {
    Shared()
}
```

The non exported variables used by the exports will be put in a new module and are imported wherever required. This is done for sharing common dependencies and removing code duplicacy.

A novel graph algorithm is used to reduce the number of shared modules. This is done keeping to only create a new module for a shared utility given that it is shared by more than one exports. This makes the code concise and more human readable.

▼ A utility file that has an exported object with many properties

```
// target.js
const nonExportedFunction = () => { ... }
const prop1 = () => {...}
const READER = {
    prop1,
    prop2: nonExportedFunction
    prop3(item) {
        READER.prop1
    }
}
export default READER

// Modified to
// target/nonExportedFunction.js
const nonExportedFunction = () => { ... }
export default nonExportedFunction

// target/prop1.js
const prop1 = () => {...}
export default prop1

// target/prop2.js
import nonExportedFunction from './nonExportedFunction'
const prop2 = nonExportedFunction
export default prop2

//target/prop3.js
import prop1FromREADER from './prop1'

const prop3 = function (item) {
    prop1FromREADER
}
```

Each individual property is considered as separate default export. The newly created modules contain the code necessary for that property. If it depends on some shared variables it imports them too. If it depends on other property of the object it imports it using an alias `propFrom<Object Name>` to avoid name collisions.

▼ A utility file with many named alias exports

```
// target.js
export {
    x,
    y as z
}

// modied to
// target/x.js
```

```
export {x}

// target/y.js
export {y as z}
```

utils-exports.json

The list of all exports of the file along with details like their exported name, export type (named | default), whether the export is an object or not are stored in a file called utils-exports.json. The default export is stored with 'default' key. This file is used later by import resolution script to modify the imports.

```
// utils.js
export const APPROVAL_STATUSES = 'hi'
const READER = {
    ...
}
export default READER
export {Hello as Call}

// utils-exports.json
{
    "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb-main/test-co
        "CREATE_UTILS_OPTION":false,
    "APPROVAL_STATUSES": {
      "exported": "APPROVAL_STATUSES",
      "type": "named",
      "obj": false
    },
    "Call": { "exported": "Hello", "type": "named", "obj": false
    "default": { "exported": "READER", "type": "default", "obj"
  }
}
```

utils-re-exports.json

Re exports from the file are like:

```
export * from './subFolder'
export {Test1 as renamed} from '../test'
```

Now such re-exports will be stored in another file called utils-re-exports.json. This file will contain all the re-exported exports from a util/index file that can be imported from that file. This also handles the recursive case when a file re-exports from another index file with its own re-exports.

```
// utils-re-exports.json
{
    "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb-main/test-c
    "renamed": {
      "imported": "Test1",
      "source": "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb
    },
    "Help": {
      "imported": "Help",
      "source": "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb
    },
    "z": {
      "imported": "z",
      "source": "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb
    },
    "Test1": {
      "imported": "Test1",
      "source": "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb
    },
    "Test2": {
      "imported": "Test2",
      "source": "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb
    }
  },
}
```

This file again will be used by import resolution script to resolve these imports

utils-types.json

Exports/Re exports of export kind as 'type' are stored in this file. To save data storage even the re-exported types are stored in this file only, in 'reExportTypes' object.

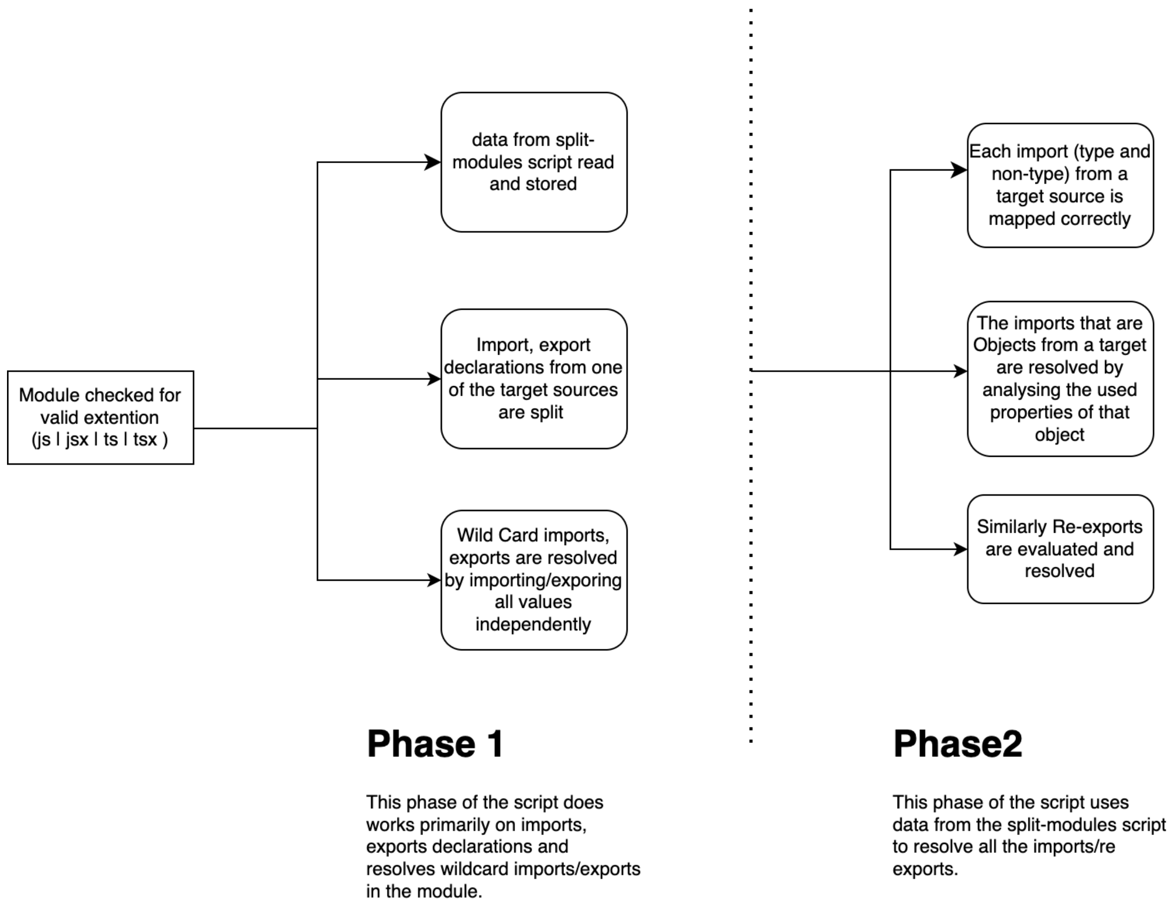
```
// types.ts
export declare type ColorPickerOverrides = {
    ColorSelectorContainer?: Override<BoxProps & Record<string,
};
export declare type ColorPickerProps = {
    overrides?: ColorPickerOverrides;
};
export declare type GradientColorResult = {
    gradientType: (typeof GRADIENT_TYPE)[keyof typeof GRADIENT_
};

export type {Color} from './colorTypes'

// utils-types.json
{
    "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb-main/barrel
    "reExportTypes": {
          "Color": {
              imported: "Color",
              source: "/Users/rhythm.aggarwal/Desktop/baseweb/ba
        },
    "ColorPickerOverrides": { "exported": "ColorPickerOverrides
    "ColorPickerProps": { "exported": "ColorPickerProps" },
    "GradientColorResult": { "exported": "GradientColorResult"
    "CREATE_UTILS_OPTION": false
  }
}
```

This data is used to map import/export of types by import-resolution script.

Import Resolution


Module checked for valid extention (js I jsx I ts I tsx)
data from split-modules script read and stored
Import, export declarations from one of the target sources are split
Wild Card imports, exports are resolved by importing/exporing all values independently
Each import (type and non-type) from a target source is mapped correctly
The imports that are Objects from a target are resolved by analysing the used properties of that object
Similarly Re-exports are evaluated and resolved
Phase 1
This phase of the script does works primarily on imports, exports declarations and resolves wildcard imports/exports in the module.
Phase2
This phase of the script uses data from the split-modules script to resolve all the imports/re exports.

Fundamental flow of the script

After the split-modules script the files `utils-exports.json` and `utils-re-exports.json` will be created that will store for each util/index files their direct exports and re-exports respectively.

The imports resolution script then parses all the modules in our entry directory and uses these files to resolve their imports.

First all the imports/re-exports with multiple specifiers are split into individual imports

```
import READER, {test as renamed} from './index'
export {x as y, z} from './index'

// modified to
import READER from './index'
import {test as renamed} from './index'
export {x as y} from './index'
export {z} from './index'
```

Then all the wildcard imports/re-exports are resolved if their source belongs to either utils-exports or utils-re-exports. These wildcard imports only contain those exports that are actually used in the code.

```
import * as say from './utils'

export * from './utils:

// modified to
import {renamed} from './utils'
import {Test} from './utils'

const say = {
    renamed: renamed,
    Test: Test
}
export { renamed} from './utils'
export {Test} from './utils'
```

Then all the individual imports/re-exports are resolved step by step.

- if an import has source in re-exports.json and the specifier is a re-exported object

```
import {renamed} from './utils'
/* where re-exports.json has
```

```
{
    "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb-main/test-c
        "renamed": {
      "imported": "Test1",
      "source": "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb
    },
    }
}
*/

import {Test1 as renamed} from './test'
```

- Then the modified inport(if modified) is checked if the source is from a exports.json. It is checked if imported specifier is an Object or not using utils-exports.json. If not then the import is simple mapped to the correct module

```
import {status as renamed} from './utils'
/* exports.json has
{
    "/Users/rhythm.aggarwal/Desktop/baseweb/baseweb-main/test-c
        "status": {
      "exported": "APPROVAL_STATUSES",
      "type": "named",
      "obj": false
    },
    }
*/

import {status as renamed} from './utils/APPROVAL_STATUSES'
```

- if its an Object, we find what all properties of the object is being used in the code and the export those only

```
import READER from '../utils';
/* exports.json
```

```
{
    "default: { "exported": "READER", "type": "default", "obj":
}
*/
export default function Main() {
    READER.accountsByChannelInfo()

    return (
        <h1>Tested{READER.aGIds}</h1>
    )
}

// modifed to
import accountsByChannelInfo from '../utils/accountsByChannelIn
import aGIds from '../utils/aGIds'
const READER = {
    aGIds: aGIds,
    accountsByChannelInfo: accountsByChannelInfo
}
export default function Main() {
    READER.accountsByChannelInfo()

    return (
        <h1>Tested{READER.aGIds}</h1>
    )
}
```

- For imports of kind ‘type’, similar steps are followed using data stored in `utils-types.json`

  with direct imports mapped to ‘./typename’ and indirect re-exported value mapped to the correct source.
- The similar modifications are done for re-exports in the module.

```
export {directExport} from './index'
export {indirectExport} from './index'
// utils-exports.json has directExport in index
// utils-re-exports.json indirectExport from './test' in index

// modified to
export {READER} from './utils/directExport'
export {indirectExport} from './test'
```