**WD401-LEVEL2**

**CONFIGURING WEBPACK:**

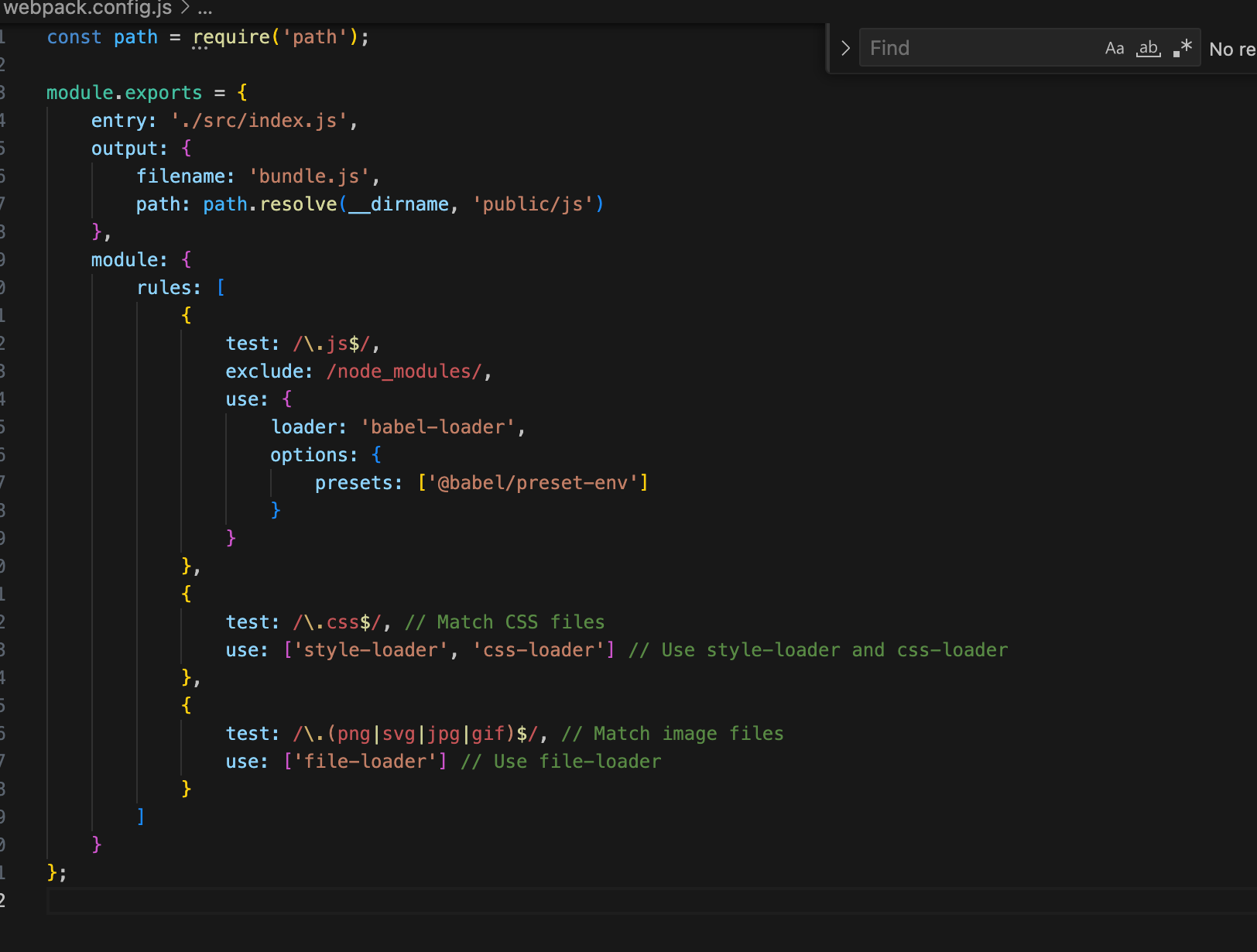
**[1.to](http://1.to) install web pack**

npm install webpack webpack-cli --save-dev

npm install babel-loader @babel/core @babel/preset-env —save-dev

**2.TO CREATE A WEBPACK CONFIGURATION FILE:**

* Create a **webpack.config.js** file in the root directory of your project.
* Configure Webpack in this file.

WEBPACK.CONFIG.JS:

**CSS and Image Handling**:

Ensure you have the necessary loaders installed:

npm install style-loader css-loader file-loader --save-dev

Now,we can import CSS and images in your javascript files and web pack will bundle them accordingly.we can create the javascript files in any directory specified in web pack.config.json

**RUN WEBPACK:**

**PACKAGE.JSON:**

"scripts": {

"build": "webpack"

}

Run **npm run build** in your terminal to bundle your JavaScript files using Webpack.

**Start express.js server:**

Ex: node index.js

**ADVANCED BUNDLING TECHNIQUES:**

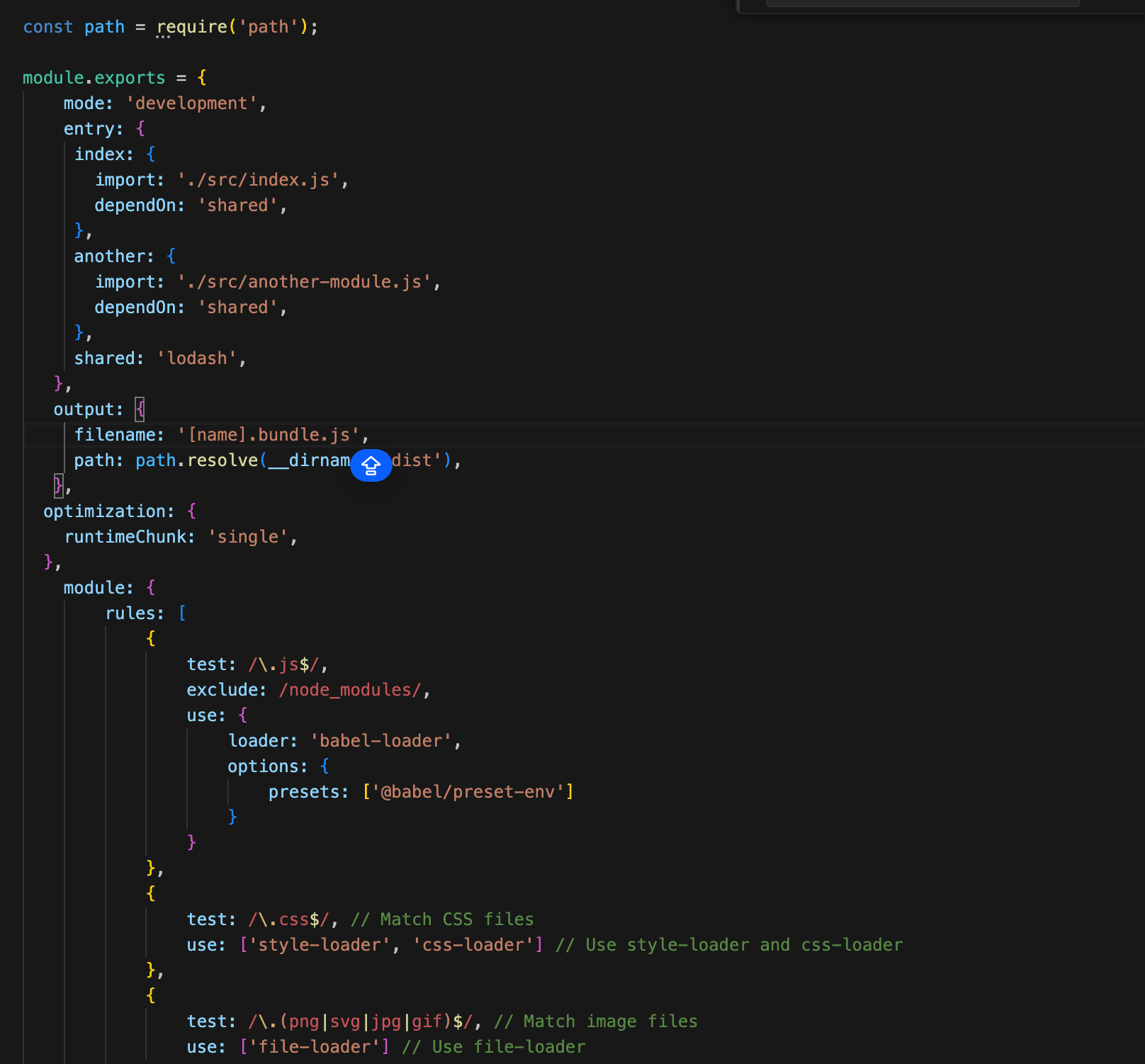
**CODESPLITTING:**

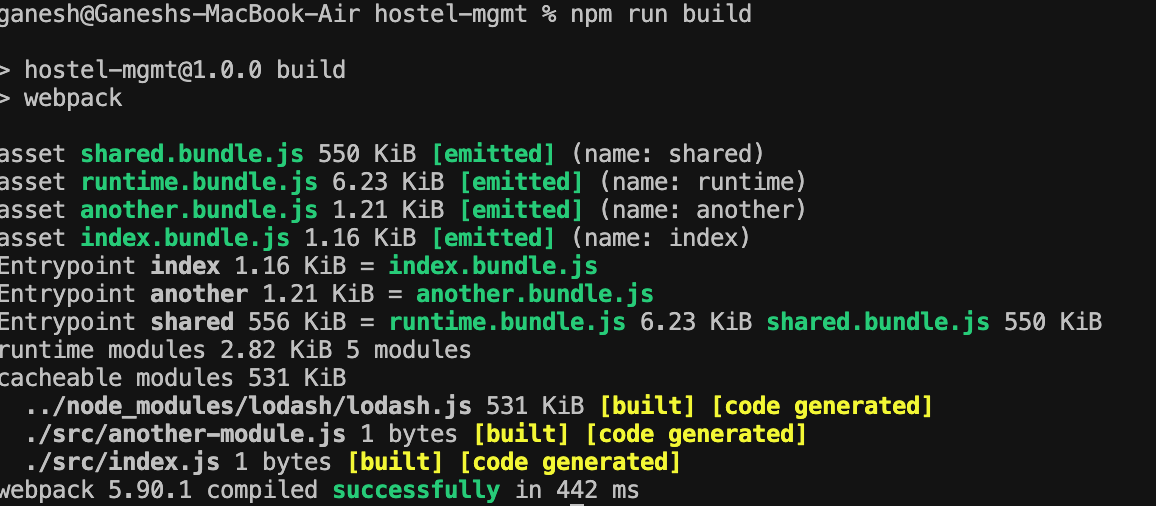
There are three general approaches to code splitting available:

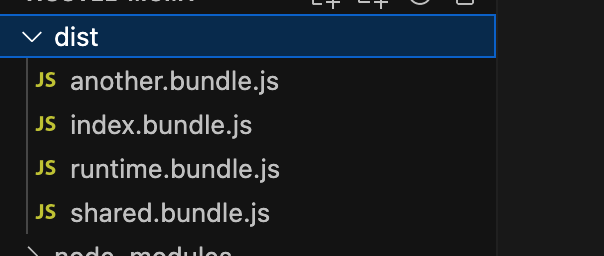
* **Entry Points**: Manually split code using [entry](https://webpack.js.org/configuration/entry-context) configuration.
* **Prevent Duplication**: Use [Entry dependencies](https://webpack.js.org/configuration/entry-context/#dependencies) or [SplitChunksPlugin](https://webpack.js.org/plugins/split-chunks-plugin/) to dedupe and split chunks.
* **Dynamic Imports**: Split code via inline function calls within modules.

WEBPACK.CONFIG.JSON:

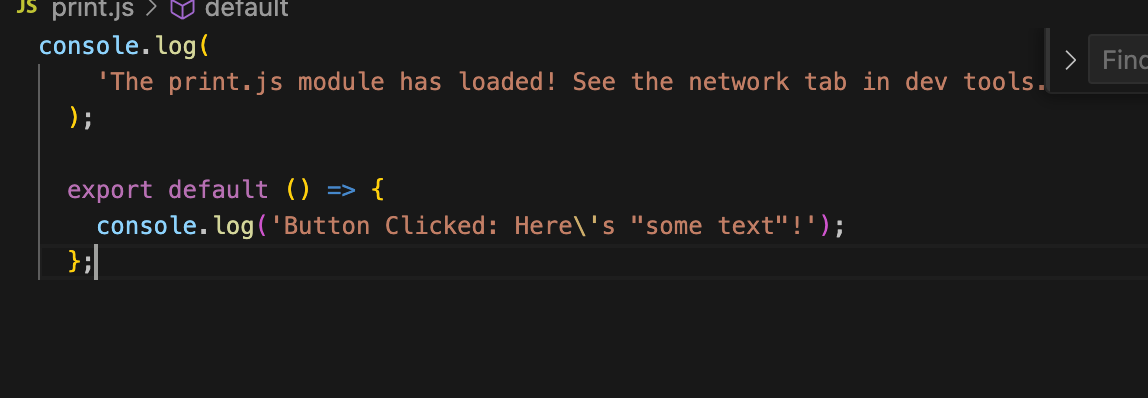
If we're going to use multiple entry points on a single HTML page, optimization.runtimeChunk: 'single' is needed too, otherwise we could get into trouble

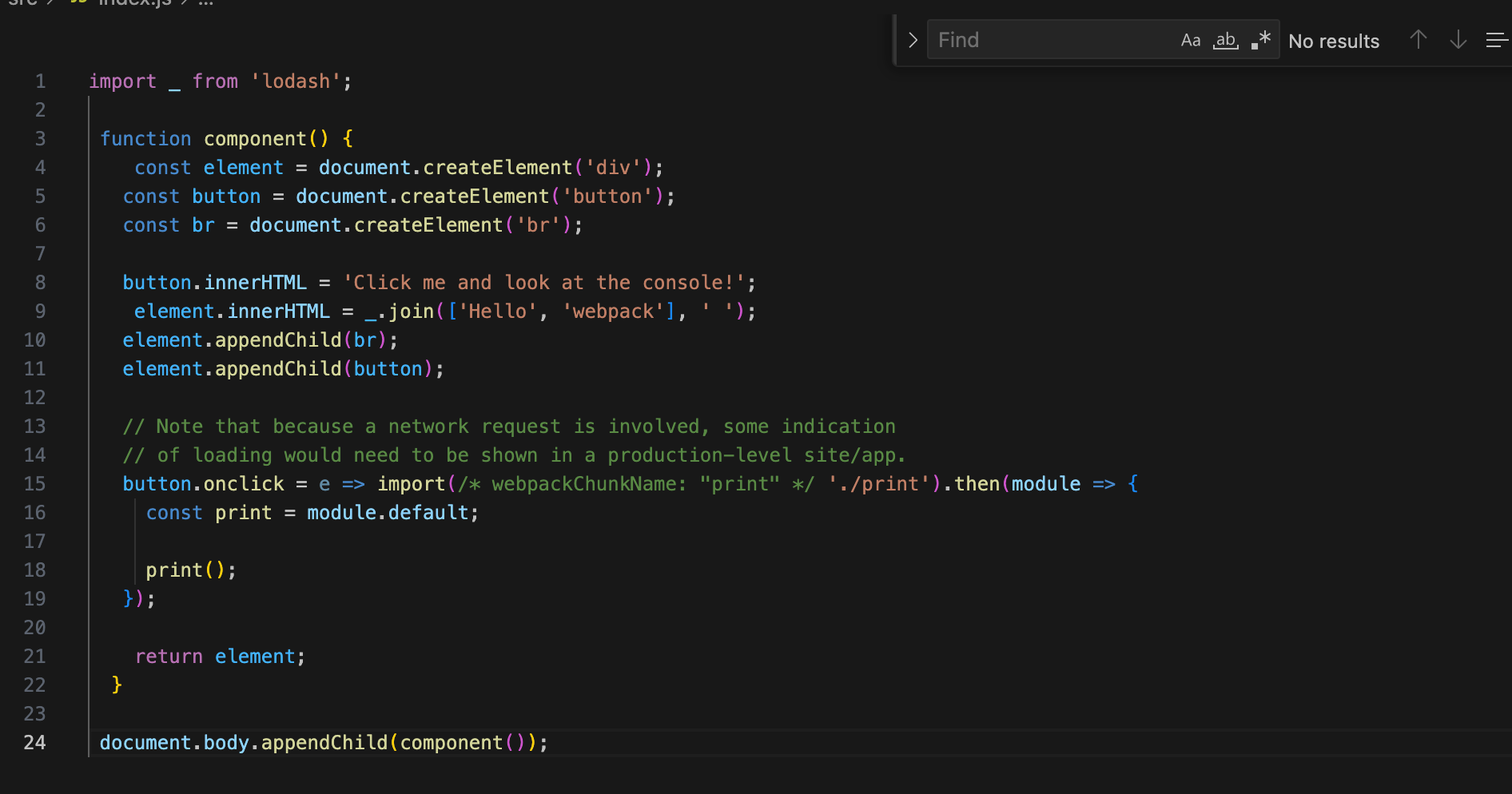


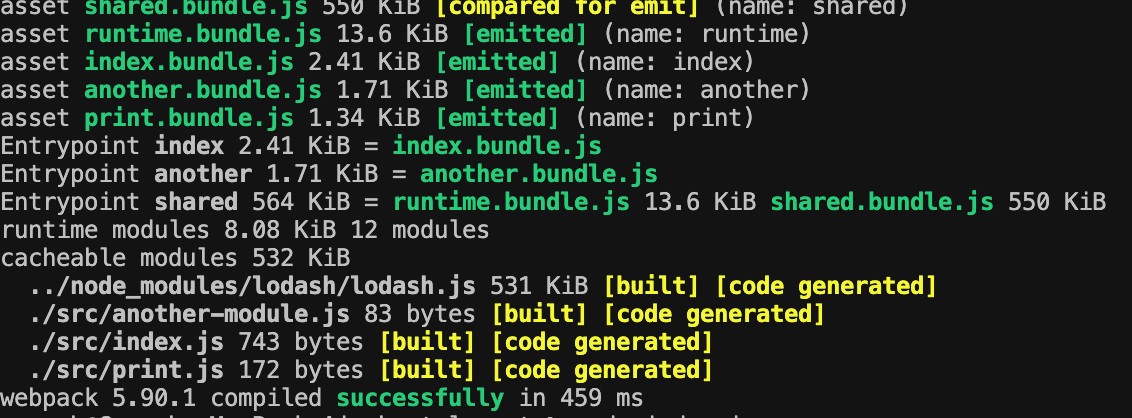
Npm run build:

FILES ARE BEING GENERATED:

**LAZY LOADING:**

**Print.js:**

Src/index.js:

Build:

**INTRODUCTION TO IMPORT MAPS:**

Import maps are a modern web standard that simplifies the management of JavaScript modules and dependencies in web applications. They provide a way to map module specifiers to actual URLs, making it easier to load and organize code.

In traditional JavaScript development, module dependencies are typically managed using import statements, which specify the path to the module being imported. However, this approach has some limitations, such as:

* **Explicit Paths**: Developers need to provide explicit paths to each imported module, which can be cumbersome and error-prone, especially in large projects with complex dependency trees.
* **Static Resolution**: Module paths are resolved statically at compile time, which means that changes to the file structure or dependency paths require updates to the import statements, leading to maintenance overhead.

Import maps address these limitations by providing a more flexible and dynamic way to manage module dependencies. An import map is a JSON file that maps module specifiers (module names) to URLs, allowing developers to define custom module resolution rules.

EX:

{

"imports": {

"lodash": "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.21/lodash.min.js"

}

}

In this import map, the module specifier **"lodash"** is mapped to the URL **"https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.21/lodash.min.js"**. Now, whenever a module imports **"lodash"**, the browser will fetch the module from the specified URL.

**Advantages of Import Maps**:

* **Simplified Module Resolution**: Import maps provide a centralized mechanism for resolving module dependencies, eliminating the need for explicit path specifications in import statements. Developers can define custom mappings for modules, making it easier to manage dependencies, especially in large projects.
* **Dynamic Module Loading**: Unlike traditional bundling approaches, which involve static resolution of module dependencies at compile time, import maps allow for dynamic loading of modules at runtime. This enables more efficient code splitting and lazy loading strategies, as modules can be fetched on-demand based on runtime conditions.
* **Better Interoperability**: Import maps promote better interoperability between different module systems and environments. They provide a standardized way to map module specifiers to URLs, making it easier to integrate modules from various sources, such as CDNs, third-party libraries, or remote servers.
* **Reduced Overhead**: Import maps can help reduce the overhead associated with bundling and transpilation processes, as they allow modules to be loaded directly from their original sources without the need for preprocessing. This can lead to faster development cycles and improved performance, particularly in development and testing environments.

**Scenarios Where Import Maps Offer Unique Benefits**:

* **Dynamic Environments**: Import maps are particularly useful in dynamic environments where module dependencies may change frequently or vary based on runtime conditions. Examples include serverless architectures, microservices, and modular web applications with complex routing requirements.
* **CDN Integration**: Import maps simplify the integration of modules from Content Delivery Networks (CDNs) by providing a standardized mapping mechanism. Developers can easily specify CDN URLs for commonly used libraries or dependencies, improving performance and reducing hosting costs.
* **Incremental Migration**: Import maps can facilitate incremental migration strategies for transitioning from traditional bundling approaches to more modular and dynamic module loading techniques. Developers can gradually adopt import maps in existing projects without the need for extensive refactoring or rewriting of import statements.

**IMPLEMENTING IMPORT-MAPS:**

**Create an Import Map File**:

Start by creating an import map file, usually named **import-map.json**. Define the mappings for your modules within this file. For example:

{

"imports": {

"lodash": "https://cdn.skypack.dev/lodash@4.17.21"

}

}

**Reference the Import Map in HTML**:

In your HTML file, reference the import map using the **<script type="importmap">** tag:

<!-- index.html -->

<html>

<head>

<script type="importmap" src="/path/to/import-map.json"></script>

</head>

<body>

<script type="module" src="/path/to/main.js"></script>

</body>

</html>

This establishes the connection between your project and the import map.

**Refactor Import Statements in JavaScript Files:**

**Refactor your existing JavaScript files to use the module specifier without specifying the full URL. For example:**

javascript:

// main.js

import \_ from 'lodash';

console.log(\_.sortBy([3, 1, 2])); // Example usage of lodash