Modularization in JavaScript: Applying Advanced Methods for Optimization

Chakradhar Avinash Devarapalli, Software Developer

E-mail: avinashd7[at]gmail.com

Abstract: *The modern digital era requires quick responses from the end users which holds the developers responsible for generating more optimized products. The modularization helps achieve this objective by enabling the developers to write more optimized and manageable code. The code splitting comes with the major advantage of faster initial page loading to save the time at user's end. However, there are certain challenges associated with the implementation of code splitting in JavaScript and it comes with difficulties in later stages after product development. This paper not only explores the role of modularization in optimization but also suggests solutions against the presented potential challenges faced by the developers during implementation. The best practices are recommended to avoid the risk of failure while carrying out code splitting in JavaScript. Some of the commonly known technologies can be used as troubleshooting methods for the effective implementation of this beneficial technique in programming.*

Keywords: *Code Splitting, Modularization, Maintainability, Reusability, Scalability, JavaScript, Optimization*

1. Introduction

With the evolution of technology, users are getting concerned about the response time of digital products. Moreover, the applications developed using JavaScript are increasing enormously with the demand for this language. With the introduction of more features, the complexity of applications has increased. Therefore, the complexity of applications and the requirement for efficiency lead the developers to adopt effective ways to optimize their code. One way to solve this problem is modularization also known as code splitting which divides the code into smaller modules that are easier to manage [1], [2].

The JavaScript allows code split to improve responsiveness in loading. When all the components are loaded in a web page for instance, the JavaScript is required to be integrated with them and executing all the JS code altogether may lead to delayed responses. Therefore, only the required JS code is loaded which is needed to make the initial page functional and later actions of users call the other code bundles. This reduces the efforts in parsing and executing JS codes during startup [3].

This research explores the role of code splitting and the problems faced by the developers during the implementation of modularization which is splitting the code into bundles and use as separate units where needed. The common problems faced during development and in later stages are difficulty in deployment, collaboration, maintainability, scalability, reusability, performance, compatibility, security, communication between modules, dependency conflicts, debugging, and testing. Effective solutions are suggested to overcome these problems including appropriate tools and best practices that can be followed during implementation.

The two main approaches to modularization are CommonJS and ES Modules. ECMAScript offers a standardized module system but React and Node.js use commonJS instead of ES modules [4] . This is because commonJS became the standard for Node.js earlier before the introduction of ES in it. The main difference comes with loading as commonJS gives synchronous and the other gives asynchronous loading [5].

Taking advantage of diverse JavaScript functionalities and utilizing the formats of CommonJS and ES from

ESMAScript, this research explores the modularization methods along with the solution to given challenges. CommonJS module format is for server-side applications and AMD is for client-side or browser-based applications. Different loaders, plugins and optimization methods are supported by these formats [5].

2. Literature Review

The JavaScript is a rich language in terms of usability and optimization which reflects in its diverse usage across platforms specifically in web based applications. It primarily offers the functionality of modularization with different environments and libraries like Node.js and React respectively. Thus this language has enormous power for completing the desired tasks [6] . Modularization refers to the division of application into smaller modules each having some functionality. This offers useful advantages like reusability and maintainability [1] . The various formats including commonJS and Asynchronous Module Definition are available in addition to ES modules for effective splitting.

The modular programming however is equipped with numerous challenges that a developer need to face from development to deployment of the system. One of the highlighted problem is that, the developers don't have access to individual files. The problem may arise when scripts are not loaded properly and order of action or appearance gets effected. The entire page may get blocked when one script or bundle is dependent on the other and the execution is blocked from one end [7] . However, the list of problems is not limited to this only and may extend to a large scope. Therefore, there is a need to overcome these challenges and to present appropriate solutions against each problem that can be occurred while code splitting.

3. Problem Statement

Although modularization offers numerous benefits to developers as well as end users with optimized products, there are certain obstacles associated with its implementation. The highlights of these problems are difficulty in dependency management, code maintenance, performance issues, challenges in scalability and compatibility, and testing difficulties. The goal is to mitigate these problems to equip the developers with appropriate solutions to face these issues. There is also a need for developers to follow suitable ways to implement the code-splitting methods in JavaScript.

4. Breaking Barriers of Code-Splitting

4.1. Deployment Problems

The dependencies are complex to handle when it comes to deployment. The multiple dependencies may raise conflicts when merged in the system altogether. This problem can be solved by employing effective collaborations and utilizing the automation tools. Further tools like Docker can be used to package the modules.

4.2. Collaboration Challenges

It is however not easy to collaborate effectively while carrying out code splitting. When code is splitted in bundles it becomes difficult to keep everyone informed with the latest changes in each bundle. This becomes more challenges when team size increases. To overcome this problem, there is a need to follow standards and maintain documentation of each update. The version control systems like Git are helpful to manage large codebase. These systems allow to not only collaborate effectively but to keep the record of changes in different versions.

4.3. Maintainability and Scalability

With the increased size of codebase, it becomes difficult for large teams to maintain and scale module based codes. The recommended way to overcome this problem to use the Single Responsibility Principle where each member of the team is responsible for maintaining the individual code and a team has to handle each bundle of the software they are working on.

4.4. Compatibility Issues

The different module formats make the system incompatible and can be challenging to run in some systems. The tools like CommonJS and AMD are needed as a source of polyfills for compatibility. The automated tools in early stages can reveal the compatibility of bundles in different environment.

4.5. Security Concerns

The use of third party libraries and dependencies can lead to the unwanted exposure of the system and ultimately raises the security concerns. However, validation techniques and effective coding can help to avoid the security issues. Moreover, regular updates and use of scanning tools are also suggested to be used over a longer run.

4.6. Cross-Modular Connection

The connection in a distributed architecture is difficult to maintain when certain code changes are implemented in the latest versions of software. Therefore, the frameworks like Redux are used for state management in largely distributed applications.

4.7. Dependency Issues

With the growth of codebase, it becomes difficult to handle conflicts between dependencies being used for the individual tasks in each module. The version conflicts may arise and make it difficult to update previous deprecated dependency or add a new dependency which has conflict with the previous one. The Node Package Manager helps to avoid this problem in association with tools like Webpack. With the use of effective package managers and tools, it becomes easier to manage these external dependencies.

4.8. Namespace Conflicts

The use of common names of modules in large systems can generate unwanted errors which are difficult to manage. Here, the naming conventions and use of assistance technologies in compilers are recommended to avoid these naming conflicts between modules.

4.9. Testing Issues

The multiple dependencies creates complexities when used for each module and make the system more robust to test each unit. The dependency injections are the only best possible way to test if the dependencies are working correctly and not deprecated.

5. Useful Methodologies

5.1. Webpack Modularization

Webpack is responsible for bundling JS files and can generate static assets for the given modules with dependencies. The different expressions that can be used for webpack are, import (ECMAScript), require () (CommonJS), define/require (Asynchronous Module Definition), imageURL (stylesheet), and @import (stylesheet) [8] . For instance, according to their official website, one way to use this is,

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
};
```

5.2. Route-Based Modularization

The modular routing helps to beat the challenges when application size increases. Routing can be effectively used in React [9] , [10]. Here is an example code of route-based code splitting of posts,

```
<Route exact path='/' component={Home} />
<Route exact path='/posts'
component={PostListing} />
<Route path='/posts/:postId'
component={PostUpdate} />
```

5.3. Dynamic Import

The dynamic import function is available to split the code on run time. To optimize the page for fast loading of components, only the modules that are required to be used are imported. For instance, the following code block shows that the code is packed in a separate module and will only be used when the user clicks the specified button. [11]

```
let button =
document.querySelector('return')

button.addEventListener('click', e => {
    return import('./tracker' )
    .then(({tracker}) => {
        tracker.getUtmParams()
    })
})
```

6. Best Practices

Apart from the suggested solutions, there are certain ways in which most of the mentioned problems can be avoided. These are,

- **Use ES6:** The dependency between the modules is managed by the import and export options of ES6. Thus, one module tends to use the other to the extent it is allowed in the other module.
- **SRP:** The Single Responsibility Principle is recommended to be followed where each module is responsible for a single activity. So, to make it easier to debug, test, and maintain the code.
- **Encapsulation and Global Scopes:** Avoid global variables and functions, instead use encapsulation with ES6 classes to prevent outer access.
- **Efficient Splitting:** Split the code into smaller chunks where bundles can be loaded easily when needed. This helps in improving initial loading time.
- **Continuous Testing:** Regularly test the individual units of the code to get only the expected result. Use tools like Cypress, Karma, and Playwright.
- **Code Documentation:** Document through different ways like manual documentation, using doc tools, or comments to later understand the code specifically in terms of code splitting.
- **Code Reviews:** Regularly review the code to ensure best practices in coding and to properly meet the project requirements.

7. Research Impact

Code Splitting positively affects both the user experience and the performance of the end product. It also helps to improve the productivity of the developer. The presented study offers solutions against the potential barriers to the implementation and effectiveness of modularization in JavaScript. It greatly impacts application optimization by highlighting the challenges of code splitting. The objective of the presented solutions and providing the best ways was to help the developer make informed decisions during the implementation of modularization in projects.

8. Conclusion

In the end, modularization or code splitting is one of the widely used techniques in modern JavaScript programming due to the vast advantages it offers in terms of code optimization and efficiency in results. As JS allows code splitting, the bundles can be separately loaded to reduce the page's initials JS payloads. This primarily became more effective after the CommonJS and ES modules.

Despite all the advantages provided by code splitting in JavaScript, there are certain obstacles faced by the developers during the process. To address these challenges, best practices can be followed by considering appropriate solutions provided in the document. More methods are evolving with time for effective modularization to help developers optimize their applications.

References

[1] S. Seydnejad, Modular Programming with JavaScript, Birmingham, UK: Pakt Publishing Ltd. , Jul. 2016.

[2] C. Mulder, "Challenges of Web Application Development: How to Optimize Client-Side Code," Aug. 2011.

[3] "Code-split JavaScript," web.dev, 04 Dec 2023. [Online]. Available: https://web.dev/learn/performance/code-split-javascript. [Accessed 2024 Mar 20].

[4] M. Mráz, "Component-based UI Web," MASARYK UNIVERSITY , 2019.

[5] K. Ubah, "CommonJS vs. ES modules in Node.js," LogRocket, 29 Dec 2021. [Online]. Available: https://blog.logrocket.com/commonjs-vs-es-modules-node-js/. [Accessed 21 Mar 2024].

[6] D. Crockford, JavaScript: The Good Parts, Sebastopol: O'Reilly Media Inc. , May. 2008.

[7] "Modularizing and Managing JavaScript," in *Modern JavaScript*, O'Reilly Media, Inc., Sep. 2015.

[8] A. Shiravanthe, "Modules and Code Splitting in Webpack 5," Medium, 20 Sep 2020. [Online]. Available: https://medium.com/@abhayshiravanthe/modules-and-code-splitting-in-webpack-5-6ce0a58d7f36. [Accessed 21 Mar 2024].

[9] K. Orjiewuru, "Cleaner and Modular Routing in React," Software Insight, Oct. 2019.

[10] A. Bijarniya, "What Is Route-Based Code Splitting in React.js?," C-sharpcorner, 05 Mar 2024. [Online]. Available: https://www.c-sharpcorner.com/article/what-is-route-based-code-splitting-in-react-js/. [Accessed 2024 Mar 22].

[11] E. Odioko, "Dynamic imports and code splitting with Next.js," 25 Aug 2022. [Online]. Available: https://blog.logrocket.com/dynamic-imports-code-splitting-next-js/. [Accessed 20 Mar 2024].