American Research Journal of Computer Science and Information Technology

Volume 7, Issue 1, 28-32 Pages
Research Article | Open Access
ISSN (Online)- 2572-2921
DOI : 10.21694/2572-2921.24006

AMERICAN RESEARCH JOURNALS
An Academic Publishing House

Asynchronous Programming in Javascript: Modern Approaches and Practice

Blahodelskyi Oleksandr Serhiyovych
Designer, MySteel BV, Gemert, Netherlands.

ABSTRACT

This work is devoted to the study of asynchronous programming in JavaScript. The article discusses modern approaches to the use of asynchrony in the development of web applications and practices in this area. Special attention is paid to asynchronous operations, promises, event handling, and other mechanisms that allow you to work effectively with asynchronous code. At the beginning of the work, the principles of asynchronous operations in JavaScript, their differences from synchronous code, as well as the role of asynchrony in modern programming are considered. The existing practices are analyzed and recommendations for users on the use of asynchronous programming to improve the performance and responsiveness of applications are identified. The research in this article is based on current research and the experience of developers, which makes it a valuable source of information for specialists interested in improving their skills in asynchronous programming in JavaScript.

KEYWORDS: *asynchronous programming, promises, web applications, async/await, callback.*

INTRODUCTION

In today's world, asynchronous programming has become an integral part of web application development, especially in JavaScript. The relevance of this topic is due to the fact that almost all useful JS programs are written using asynchronous development methods. To understand the importance of the asynchronous approach, it is essential to grasp the fundamental differences between synchronous and asynchronous code, along with their advantages and disadvantages.

Synchronous code implies a pipeline nature where the execution of subsequent parts is delayed until the previous part completes, thus executing code sequentially, line by line. This method is suitable for performing the simplest and most primitive tasks where the speed of operation is not critical. Consequently, synchronous programming is not ideal for working with large files or making server requests where high application responsiveness is crucial. In synchronous code, if one of the operations takes a long time, the entire workflow will be paused until it completes, leading to low application responsiveness.

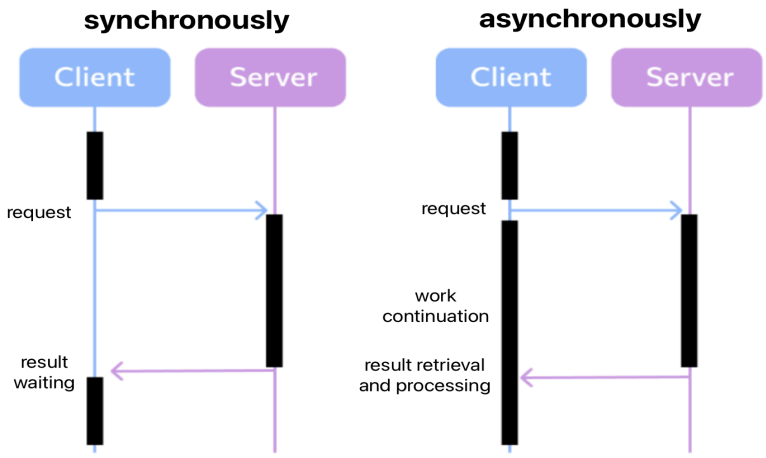

Fig. 1. Schematic comparison of synchronous and asynchronous approaches.

In asynchronous code, operations can be performed simultaneously without blocking the rest of the program, differing from the synchronous approach. This is particularly useful for time-consuming operations, such as network requests or input-output operations. Asynchronous programming allows such operations to run while continuing to execute other tasks until the initial operation completes.

In the modern IT sphere, JavaScript is used for developing and maintaining user web applications where interface responsiveness is crucial [1]. The practice of asynchronous programming enables operations to run in parallel, avoiding the blockage of the main function of the application. Asynchronous programming is indispensable when working with server requests, as the speed of operations increases the application's throughput.

EVOLUTION OF ASYNCHRONOUS METHODS AND MODERN APPROACHES TO THEIR IMPLEMENTATION

Over the past decade, asynchronous programming in JavaScript has exhibited significant diversity. From basic callbacks to the use of promises and the async/await constructs, each of these tools was created to address specific challenges [6].

Callback

Callbacks represent a fundamental approach to asynchronous programming in JavaScript. A callback is a function that is passed to another function as an argument and is executed after the completion of an asynchronous operation [4]. This method allows certain actions to be performed after the operation finishes without blocking the main execution thread.

Server requests, events, and timers are all asynchronous operations that utilize callbacks for effective operation.

Table 1. Analysis of the Advantages and Disadvantages of Callbacks

Advantages	Disadvantages
Simplicity: callbacks are easy to learn and easy to use.	Callback Hell: If more callbacks are applied, the code may break or become hard to read for the user.
Flexibility: Callbacks are used in a wide variety of situations, from complex requests to the server to simple timers.	Errors and debugging: Sometimes there are difficulties with error handling in callback functions, especially in case of complex asynchronous operations.

The main issue with using callbacks in modern asynchronous programming is known as “Callback Hell” or the “Pyramid of Doom.” This phenomenon occurs when there is an excessive layering of nested callbacks, making the code difficult to read, debug, and maintain.

Promises

The use of promises marked an important step in the evolution of asynchronous programming in JavaScript, providing a more convenient and structured approach to managing asynchronous operations compared to callbacks. A promise is an object that represents the eventual completion (or failure) of an asynchronous operation [2]. The Promise constructor is used to create a promise, which accepts a function with two parameters: resolve and reject. These parameters are used to conclude the promise successfully or with an error, respectively.

Example of a promise:

```
let promise = new Promise(function(resolve, reject) {
  // Asynchronous operation
  if (/* success */) {
    resolve(result);
  } else {
    reject(error);
  }
});
```

Async/Await

The introduction of the async and await keywords in JavaScript significantly simplified working with asynchronous code, making it more understandable and readable [3, 7]. This approach allows writing asynchronous code that looks and behaves like synchronous code, improving the overall program structure.

The async keyword is used to declare a function as asynchronous. Such a function always returns a promise. If the function returns a value, it is automatically wrapped in a promise that resolves with that value. If an error occurs, the function returns a promise that rejects with that error.

Example of async function:

```
async function fetchData() {
  return “Data”;
}
```

The await keyword is used inside asynchronous functions to wait for the completion of a promise [7]. When awaiting a promise, the execution of the function is paused and then resumed after the promise is resolved or rejected, returning the result of the operation as the promise’s value. If the promise is rejected, await throws an exception that can be handled using a try/catch block.

Example of using async/await:

```
async function fetchData() {
  try {
    let response = await fetch(‘https://api.example.com/data’);
    let data = await response.json();
    return data;
  } catch (error) {
    console.error(‘Error:’, error);
  }
}
```

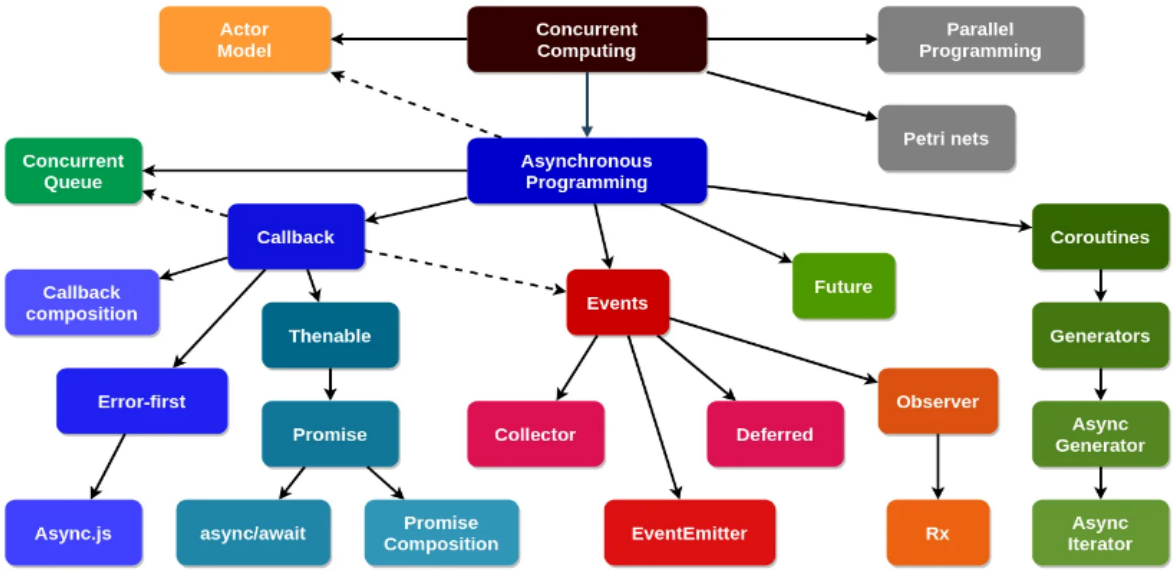

Fig. 2. Various Ways of Working with Asynchrony

This diagram shows different ways of working with asynchronous code. The colored blocks pertain to asynchronous programming, while the black-and-white blocks represent methods of parallel programming and Petri nets, which, like asynchronous programming and the actor model, represent different approaches to parallel computing. Events and concurrent queues are connected to callbacks through dashed lines because these concepts are based on the use of callbacks, although they represent significantly newer methods of operation.

PRACTICES OF ASYNCHRONOUS CODE

1) Error Handling with try/catch

In asynchronous programming, proper error handling is particularly important. Consider the example of using the try/catch construct:

```
async function fetchUserData(userId) {
 try {
  const response = await fetch(`https://api.example.com/users/${userId}`);
  if (!response.ok) {
   throw new Error(`HTTP error! status: ${response.status}`);
  }
  const userData = await response.json();
  return userData;
 } catch (error) {
   console.error(`Error fetching user data: ${error.message}`);
  throw error; // Throw the error for processing at a higher level
 }
}
```

This approach allows localizing error handling directly where the errors occur, facilitating more effective debugging and code maintenance.

2) Breaking Down Asynchronous Logic into Small Functions

Decomposing complex asynchronous operations into simpler functions improves code readability and testability:

```
async function getUserProfile(userId) {
 const userData = await fetchUserData(userId);
 const userPosts = await fetchUserPosts(userId);
 return {
  ...userData,
  posts: userPosts
 };
}
async function fetchUserData(userId) {
 // Realization of user data request
}
async function fetchUserPosts(userId) {
 // Implementation of querying user's posts
}
```

This division makes understanding the program logic easier and simplifies unit testing of individual components.

3) Using Promise.all for Parallel Execution of Tasks

To optimize performance when dealing with multiple asynchronous operations, it is advisable to use the Promise.all method:

```
async function fetchMultipleUserProfiles(userIds) {
   const profilePromises = userIds.map(userId => getUserProfile(userId));
 return await Promise.all(profilePromises);
}
```

This method allows several asynchronous operations to be

executed in parallel, significantly reducing the total execution time.

4) Documenting Asynchronous Functions

High-quality documentation is critical for code maintenance and development. Example of documenting an asynchronous function:

```
/**
 * Asynchronously retrieves a user's profile.
 * @param {number} userId - User ID.
 * @returns {Promise<Object>} An object that contains user profile data.
 * @throws {Error} If an error occurs while receiving data.
 */
async function getUserProfile(userId) {
  // Function realization
}
```

This approach to documentation facilitates understanding the purpose of the function, its parameters, and return values.

5) Choosing an Asynchronous Programming Approach

In modern development, the most common methods are promises and async/await. Comparing their usage:

```
// Use of promises
function fetchDataPromise() {
  return fetch('https://api.example.com/data')
    .then(response => response.json())
    .catch(error => console.error('Error:', error));
}

// Using async/await
async function fetchDataAsync() {
  try {
    const response = await fetch('https://api.example.com/data');
    return await response.json();
  } catch (error) {
    console.error('Error:', error);
  }
}
```

The choice between these approaches depends on the specific project requirements and the developer's personal preferences. Async/await provides more linear and readable code, especially in complex asynchronous scenarios.

The presented examples demonstrate key aspects of modern asynchronous programming in JavaScript, ensuring code efficiency, readability, and reliability [5].

CONCLUSION

Modern web development in JavaScript requires developers to have a deep understanding of asynchronous programming and the ability to effectively apply its principles. Based on the studied principles, developers can create more reliable, scalable, and high-performance web applications. It is important to properly structure the code, optimize data processing, and ensure the seamless operation of the application.

Asynchronous programming in JavaScript plays a crucial role in modern web development, allowing the creation of dynamic and interactive applications. By following best practices, using modern tools, and continually improving their skills, developers can achieve success and create high-quality software products.

While working on this scientific article, it became evident that the topic of asynchronous programming in JavaScript is so vast and multifaceted that it deserves a more in-depth and detailed examination. This realization led to the idea of creating a full-fledged book dedicated to this topic. This article served as a starting point for this larger project, laying the foundation for further research and analysis of modern approaches to asynchronous programming in the context of web development.

The planned book aims to provide developers with a comprehensive guide to the effective use of asynchronous techniques in JavaScript, covering both fundamental concepts and advanced strategies for their application in real-world projects. It will serve as a logical continuation and expansion of the ideas presented in this article, offering readers an in-depth understanding of this critically important area of modern programming.

REFERENCES

1. Simpson K. You Don't Know JS Yet: Scope & Closures. 2020. Available from: https://github.com/getify/You-Dont-Know-JS
2. CSS-Tricks. Understanding JavaScript's async await. 2020. Available from: https://css-tricks.com/understanding-async-await/
3. Tominaga E., Arahori Y., Gondow K. AwaitViz: a visualizer of JavaScript's async/await execution order //Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. – 2019. – pp. 2515-2524.
4. Udemy. Asynchronous JavaScript: Promises, Callbacks, Async/Await. 2021. Available from: https://www.udemy.com/course/asynchronous-javascript-promises-callbacks-async-await/
5. Coursera. JavaScript, jQuery, and JSON. 2020. Available from: https://www.coursera.org/learn/javascript-jquery-json

6. Flanagan D. JavaScript: The Definitive Guide, 7th Edition. 2020.

7. Medium. JavaScript Async/Await: Making Asynchronous Programming Simpler. 2021. Available from: https://medium.com/javascript-in-plain-english/javascript-async-await-making-asynchronous-programming-simpler-7a8b0c7d64f7

Citation: Blahodelskyi Oleksandr Serhiyovych, "Asynchronous Programming in Javascript: Modern Approaches and Practice", American Research Journal of Computer Science and Information Technology, Vol 7, no. 1, 2024, pp. 28-32.