International Journal of Computer (IJC)

ISSN 2307-4523 (Print & Online)

https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

Typing in JavaScript API SDK development: Benefits and Implementation Techniques Using TypeScript

Lyamkin Ilya*

Senior Full Stack Engineer at Spotify,USA

Email: ilya.lyamkin@gmail.com

Abstract

This article aims to explore the development of a scalable and maintainable API SDK using TypeScript, with a focus on the practical implementation of modern programming techniques. The study presents a detailed methodology, including the selection of TypeScript for strict type enforcement, the use of Rollup and microbundle for optimized bundling, and the application of modular design principles through TypeScript mixins. The results highlight the advantages of these approaches in creating a lightweight, cross-platform SDK that works seamlessly in both browser and Node.js environments. Testing strategies, including the use of Nock for HTTP request simulation, are also discussed to ensure reliability and stability. The conclusions emphasize the significance of these modern practices in enhancing code quality, maintainability, and scalability. The novelty of this work lies in its comprehensive integration of these methodologies, providing a robust framework for API SDK development in contemporary software engineering.

Keywords: typescript; api sdk; modular design; rollup; microbundle; mixins; cross-platform; nock; software development; testing.

1. Introduction

In recent years, the Dev.to API has become an essential tool for developers looking to integrate the platform's functionalities into their applications [1]. However, developing client libraries to interact with this API presents a number of challenges. These include the need to ensure reliable authentication, correctly define data models, and create API controllers that must remain up-to-date as the API evolves. These issues are critical for any developer aiming to build a sustainable and maintainable SDK.DevtoJS [2], an open-source library written in TypeScript, offers a solution to many of these problems. This library is designed to simplify interactions with the Dev.to API by providing developers with tools that are both easy to use and efficient.

Received: 9/1/2024
Accepted: 11/1/2024
Published: 11/11/2024

----

* Corresponding author.

DevtoJS allows developers to quickly integrate the Dev.to API into their projects while minimizing the challenges associated with SDK maintenance and updates.One of the main tasks developers face when working with the Dev.to API is creating reliable authentication, defining and normalizing data models, and implementing flexible and powerful API controllers. It is crucial that the SDK remains current and adaptable to changes in the API [3]. In this context, TypeScript proves to be an important tool that enhances the development process. It provides strict typing, which significantly reduces errors and simplifies code maintenance, especially when dealing with API changes.

In recent years, the use of TypeScript in API SDK development has become a subject of active study and discussion within the developer community. Research has shown that strict typing in TypeScript significantly improves code quality and reduces the occurrence of errors during development. For instance, a study conducted on over 600 projects using both JavaScript and TypeScript revealed that TypeScript projects demonstrate higher code quality, better cognitive complexity, and fewer "code smells" compared to JavaScript projects. However, this does not always translate to fewer bugs or faster bug resolution times, highlighting the complexity of these processes even when using strict typing [4].

Other studies emphasize the importance of automation in TypeScript SDK development. For example, the dts-generate tool allows for the automatic generation of TypeScript declaration files from existing JavaScript libraries, which helps reduce the number of errors during integration with external libraries and accelerates the migration to TypeScript [5]. This is especially important in the context of SDKs, which often require frequent adaptation to evolving APIs.

Moreover, the handling of types in TypeScript continues to evolve. Research shows that reducing the use of the any type in TypeScript projects positively impacts code quality and understandability, though it requires greater attention from developers in defining and maintaining types [4]. However, the development and use of tools like FlexType automate this process, significantly reducing the time required to define types and improving code performance [6].

Thus, modern research confirms that the use of TypeScript in SDK development not only enhances code reliability and predictability but also requires careful type configuration and automation, particularly when dealing with dynamic APIs.

2. Advantages of typing in API SDK development

Typing, introduced into the API SDK development process through TypeScript, represents a fundamental improvement in programming, especially when dealing with dynamic and evolving APIs like Dev.to. In this context, typing becomes more than just a tool; it is a crucial conceptual element that profoundly impacts the entire software architecture.

Let's consider the aspect of defining and normalizing data models. In traditional JavaScript development, data retrieved from APIs is handled without explicit typing, which can lead to runtime errors, particularly when integrating complex and multi-layered objects [7]. TypeScript addresses this issue by introducing static typing,

allowing developers not only to define data structures at the level of interfaces and types but also to ensure their consistency throughout the data processing stages [8]. This significantly reduces the likelihood of errors, as any type mismatch is detected at the compilation stage. As a result, developers gain confidence that their code will function correctly, even when handling complex data formats or passing data through multiple processing layers.This strict typing contributes to improved code readability and maintainability. Developers can clearly understand the types of data they are working with in each case, making the code more transparent and easier to comprehend. This is especially important in team-based development, where the code passes through the hands of multiple developers. In TypeScript, when a developer sees an interface type, they know exactly what data to expect and can be confident that this data will be handled correctly. In contrast, JavaScript, with its dynamic typing, lacks such guarantees, forcing developers to rely on documentation or comments in the code, which is not always reliable [9].

Moreover, strict typing allows for efficient management of changes in the API. When developing an SDK for dynamic APIs like Dev.to, developers inevitably face the need to update their code in response to API changes. With TypeScript, this task becomes significantly simpler, as any changes in data types or API methods are immediately reflected in the SDK's interfaces and classes. For example, if a new field is added to the API, it can be incorporated into the corresponding interface, and TypeScript will automatically ensure that all methods and functions using that interface are correctly updated. This not only speeds up the adaptation process but also reduces the risk of errors caused by overlooked or improperly updated code sections.

To illustrate the advantages of using TypeScript in defining and normalizing data models, consider the following code example:

```
interface Article {
  id: number;
  title: string;
  description: string;
  body_markdown: string;
  published_at: Date;
  author: Author;
  tags?: string[]; // New field that can be easily added
}

interface Author {
  name: string;
  username: string;
  bio?: string; // Additional optional field
}
```

Figure 1

This code demonstrates how easily data models can be managed using TypeScript interfaces. If the Dev.to API

adds a new field, it can be easily added to the interface, and the entire system automatically adapts to this change. The programmer receives immediate notifications about the need to update other parts of the code using this interface.Enhancing control over API methods and their usage is another important aspect that receives significant improvements through TypeScript. In API SDK development, it is crucial not only to correctly define the methods for interacting with the API but also to ensure their proper use throughout the project. TypeScript allows developers to achieve this through strict typing of method arguments and return values. Developers can be confident that correct data is passed when calling an API method, and the results of this call are handled properly. For example, if an API method returns an array of `Article` objects, TypeScript guarantees that these objects will have the correct structure, and any attempt to access non-existent fields will result in an error at compile time rather than runtime.

A diagram illustrating the type-checking process at compile time can be useful for understanding how TypeScript prevents errors before code execution:

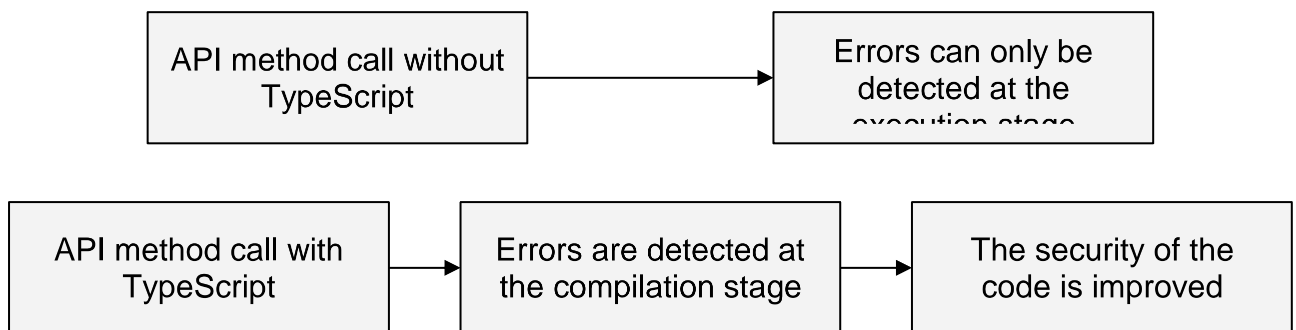

Figure 2: Type-checking process at compile time

Additionally, strict typing simplifies the implementation of polymorphism and encapsulation in code. With TypeScript, developers can create abstractions that hide complex implementation details while providing a simple and intuitive interface for interacting with the API. For example, it is possible to create a controller class that encapsulates all operations with data of a specific type, ensuring that these operations are performed safely and correctly. This not only improves the structure and organization of the code but also makes it more flexible and scalable.

At the level of SDK updates and maintenance, TypeScript proves to be an indispensable tool. When the API changes, developers face the challenge of quickly and safely updating the SDK to keep it current. In JavaScript, where there is no typing, such a task can become a real challenge since errors might only be discovered at runtime. With TypeScript, however, the update process becomes much simpler: changes in the API are reflected in the types and interfaces, allowing the SDK to be quickly adapted with the assurance of its proper functionality [10].

Let's consider how the interface and API methods change when a new field is added to the API. Below is a code example that demonstrates how TypeScript allows for a swift adaptation of the SDK to changes:

```
interface Article {
  id: number;
  title: string;
  description: string;
  body_markdown: string;
  published_at: Date;
  author: Author;
  tags?: string[]; // New field
}

class DevToAPI {
  getArticles(): Promise<Article[]> {
    return axios.get('/articles').then(response => response.data as Article[]);
  }

  // New method using the new field
  getTaggedArticles(tag: string): Promise<Article[]> {
    return axios.get(`/articles?tag=${tag}`).then(response => response.data as
Article[]);
  }
}
```

Figure 3

Thus, using TypeScript in API SDK development significantly enhances all aspects of the process, from defining and normalizing data models to controlling API methods and usage, as well as simplifying SDK updates and maintenance. TypeScript provides developers with a powerful tool for creating reliable, secure, and scalable code, which is especially important when working with dynamic APIs like Dev.to.

3. Typing implementation methods in API SDK using DevtoJS as an example

To implement an efficient and scalable API SDK, such as DevtoJS, it is essential to follow several key steps that ensure high-quality and long-lasting software [11].The development begins with selecting the programming language and target platform. As discussed earlier, TypeScript is the optimal choice for creating SDKs due to its strict typing and support for modern development practices. TypeScript enables developers to write code that functions correctly in both the browser and Node.js environments, which is crucial when creating universal libraries.

The next step involves packaging the library. Minimizing size and ensuring support for different browser versions are key tasks that must be addressed at this stage. Using Rollup in combination with microbundle helps

efficiently manage the build process. Rollup, as a bundling tool, allows for the creation of compact bundles, while microbundle simplifies the setup and automatically generates multiple versions of the library for different environments, such as CommonJS, ES Modules, and UMD. An example configuration for microbundle:

```
{
  "source": "src/index.ts",
  "main": "dist/index.js",
  "module": "dist/index.module.js",
  "unpkg": "dist/index.umd.js",
  "scripts": {
    "build": "microbundle",
    "dev": "microbundle watch"
  }
}
```

Figure 4

To ensure the scalability of the library, it is important to carefully design its structure. Dividing the code into modules, each responsible for a separate API resource, makes it easy to extend functionality as needed. For example, in the case of DevtoJS, the code can be organized so that each resource (articles, comments, users) is represented by a separate module, with all modules later combined into a main class using mixins. This approach helps maintain clean and readable code while ensuring flexibility and scalability.

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            Object.defineProperty(
                derivedCtor.prototype,
                name,
                Object.getOwnPropertyDescriptor(baseCtor.prototype, name)
            );
        });
    });
}

class DevTo extends Base {}
interface DevTo extends Articles, Comments, Users {}
applyMixins(DevTo, [Articles, Comments, Users]);
```

Figure 5

In this context, mixins allow the combination of functionality from various modules without disrupting the

logical structure of the library. This approach aligns with object-oriented programming principles and helps avoid code duplication.

When developing an API SDK, special attention should be paid to implementing universal requests that need to work across various environments. The `isomorphic-unfetch` library addresses this issue by providing a unified interface for making HTTP requests in both browser and Node.js environments. This approach simplifies development and makes the code more resilient and portable.

```
request<T>(endpoint: string, options?: RequestInit): Promise<T> {
  const url = this.basePath + endpoint;
  const headers = {
    'api-key': this.apiKey,
    'Content-type': 'application/json',
  };
  const config = {
    ...options,
    headers,
  };
  return fetch(url, config).then(r => {
    if (r.ok) {
      return r.json();
    }
    throw new Error(r.statusText);
  });
}
```

Figure 6

This code fragment demonstrates how to implement a request function that can be used in both client and server applications. This approach ensures the universality and portability of the code, which is critically important for creating a high-quality SDK.Testing plays a key role in the SDK development process. To ensure the correctness of the code, particularly when interacting with external APIs, tools such as Nock should be used. Nock allows developers to simulate HTTP requests and verify their accuracy without the need to interact with the actual API. This significantly speeds up the development process and increases code reliability.

```
describe('Article resource', () => {
  test('getArticles returns a list of articles', async () => {
    const scope = nock('https://dev.to/api/')
      .get('/articles')
      .reply(200, [{ title: 'Article' }]);

    const DevToClient = new DevTo({ apiKey: 'XYZ' });
    const articles = await DevToClient.getArticles();

    expect(articles).toEqual([{ title: 'Article' }]);
    scope.done();
  });
});
```

Figure 7

This test demonstrates how Nock can be effectively used to verify the correctness of requests and responses, which is especially important in integration testing. Ensuring that the SDK functions correctly regardless of API changes is crucial to successful development.Thus, creating an API SDK requires thorough attention at each stage—from tool selection to structure organization and testing. Using modern approaches such as strict typing, modularity, and universal requests helps create a library that is not only functional but also easy to maintain, scalable, and resilient to changes. This approach ensures that the SDK meets high standards of quality and performance.

4. Discussion and Limitations

The development of an SDK for an API using TypeScript, as discussed in the article, certainly opens up numerous possibilities for improving code quality and predictability. However, it is important to address certain aspects that may not be immediately apparent during development but can significantly influence the final outcomes.

One of the key advantages of TypeScript, which has already been covered, is its strict typing. However, in real-world projects, we often face situations where strict typing can be both a benefit and a burden. In particular, working with rapidly changing APIs, as is the case with Dev.to, clearly benefits from strict typing, but on the other hand, every change in the API's structure requires corresponding updates to the types, which is not always a smooth process. There are scenarios where the data cannot be precisely defined at compile-time, forcing developers to use more flexible types like any, which undermines the advantages of strict typing and reduces confidence in the correctness of the code. Moreover, developers sometimes spend more time configuring and debugging types than they do working on the core business logic.

Additionally, the issue of evolving APIs brings up another important aspect — the need for regular SDK updates. Strict typing simplifies this process by surfacing errors and inconsistencies at the compilation stage, but at the same time, it demands constant monitoring of API changes and quick adaptation of types and interfaces. Otherwise, types can start to diverge from the actual data, leading to the accumulation of technical debt. In large projects, this becomes especially relevant as maintaining the SDK's accuracy requires significant resources during refactoring phases.Regarding modularity and the use of mixins, the situation is similar. The modular design principle undoubtedly helps structure the code and makes it easier to extend, but it also introduces a downside: such a structure demands strict control over dependencies. Otherwise, the code may become overly fragmented, complicating its maintenance. While mixins provide convenience, they can obscure implementation details, creating the illusion of simplicity. In reality, overusing mixins can lead to a convoluted and hard-to-debug system, especially when dealing with complex interactions between modules. Managing the complexity of such an architecture requires not only technical expertise but also meticulous documentation, which may not always be feasible under tight development deadlines.Testing is another crucial point worth discussing. While using Nock to simulate requests indeed simplifies the testing process, it does not cover all possible scenarios. API simulations cannot fully replicate real-world conditions, especially when it comes to network failures or edge cases. As a result, even well-tested SDKs may fail in production environments if the tests did not account for specific scenarios. This limitation is not unique to TypeScript or Nock, but it is essential to acknowledge when developing complex systems where the reliability of the API SDK is paramount.Furthermore, it's important to recognize that TypeScript introduces a high learning curve, particularly for developers unfamiliar with the tool. Despite its clear benefits, TypeScript requires time for teams to learn and adopt new concepts. In a fast-paced development environment, this can act as an additional barrier. Some developers may find the transition to TypeScript unnecessarily complex, especially if they are accustomed to working with dynamic JavaScript. This should also be considered a limitation, as the initial speed of development may slow down while the team becomes familiar with the new tools and methodologies.Ultimately, while the proposed approaches certainly provide a robust framework for SDK development, they require constant attention to detail, particularly during maintenance and updates. Strict typing and modularity are powerful tools, but their effectiveness depends directly on their thoughtful application and a deep understanding of the principles behind them, both within the team and when integrating with external systems.

5. Conclusion

This article provided a comprehensive analysis of API SDK development, using the DevtoJS library as an example, with a focus on leveraging TypeScript as the primary tool for enforcing strict typing, improving code structure, and enhancing reliability. We explored key development stages, from selecting the language and build tools to structuring the code and testing.The main emphasis was on how TypeScript's strict typing minimizes errors and improves code readability, which is crucial in the context of developing scalable SDKs. Proper code structuring, based on modularity and the use of mixins, was shown to create flexible and easily extendable libraries. We also discussed approaches to ensuring cross-platform compatibility and minimizing library size, making it more versatile and convenient for various projects.However, as discussed in the article, strict typing and modular design impose certain limitations, such as the need for frequent updates to types in response to API changes and increased demands for maintaining the code's architecture. Additionally, the use of tools like Nock

highlighted the importance of integration testing to maintain high code reliability when interacting with external APIs, though it is not always possible to fully simulate all real-world scenarios.Research confirms that TypeScript improves code quality, but it also requires additional effort to automate and maintain type definitions. Modern tools like dts-generate and FlexType help automate this process, speeding up library migration and improving work with dynamic APIs.In conclusion, using modern tools and methodologies such as TypeScript, Rollup, microbundle, and Nock enables the creation of API SDKs that meet high standards for quality, maintainability, and performance. By following the recommendations presented in this article, developers can build high-quality SDKs capable of working efficiently across various environments, while addressing the limitations and challenges these projects often encounter.

References

[1]. Dev community. URL: https://dev.to/

[2]. DevtoJS. URL: https://github.com/ilyamkin/dev-to-js

[3]. How to Build a Dev.to API Client Library in JS. URL: https://dev.to/ilyamkin/how-to-build-a-dev-to-api-client-library-in-js-3i1j

[4]. Bogner J., Merkel M. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github //Proceedings of the 19th International Conference on Mining Software Repositories. – 2022. – C. 658-669.

[5]. Cristiani F., Thiemann P. Generation of typescript declaration files from javascript code //Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes. – 2021. – C. 97-112.

[6]. Voruganti S., Jesse K., Devanbu P. FlexType: a plug-and-play framework for type inference models //Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. – 2022. – C. 1-5.

[7]. Thennakoon R., Hettige B. A STUDY ON OBJECT-ORIENTED DESIGN PRINCIPLES AND PATTERNS. – 2022.

[8]. Matuszek D. Quick JavaScript. – Chapman and Hall/CRC, 2023.

[9]. Bierman G., Abadi M., Torgersen M. Understanding typescript //ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28. – Springer Berlin Heidelberg, 2014. – C. 257-281.

[10]. Jansen R. H. Learning TypeScript 2. x: Develop and maintain captivating web applications with ease. – Packt Publishing Ltd, 2018.

[11]. Vanderkam D. Effective TypeScript. – " O'Reilly Media, Inc.", 2024.