**ISSN:** 2583-9888
**DOI:** doi.org/10.51219/JAIMLD/chakradhar-avinash-devarapalli/62

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 1 & Iss: 1 *Review*

# Handling External API Challenges for Bridging Frontend Applications

**Chakradhar Avinash Devarapalli***

Software Developer, USA

**Citation:** Chakradhar Avinash Devarapalli. Handling External API Challenges for Bridging Frontend Applications. *J Artif Intell Mach Learn & Data Sci 2022*, 1(1), 165-169. DOI: doi.org/10.51219/JAIMLD/chakradhar-avinash-devarapalli/62

**Received:** December 03, 2022; **Accepted:** December 28, 2022; **Published:** December 30, 2022

***Corresponding author:** Chakradhar Avinash Devarapalli, Software Developer, USA, E-mail: avinashd7@gmail.com

## A B S T R A C T

API integration poses challenges in frontend development. Issues include data format compatibility, inadequate documentation, security risks, legacy system integration, scalability concerns, versioning issues, and error handling. Solutions involve standardizing formats, enhancing documentation, bolstering security, modernizing systems, ensuring scalability, managing versions, and handling errors effectively. The Axis Bank case study illustrates these solutions. Adapting to evolving tech is crucial for meeting digital consumer needs.

**Keywords:** API integration, Frontend developer, Challenges, Solutions

## 1. Introduction

Web and application development has been a constantly evolving niche since the 2000's. Especially now, as people become more inclined to make purchase decisions not just because of the price, but also because of the experience they get, developers are under immense pressure to juggle better results, improved efficiency, and quicker turnaround.

In such an environment, Application Programming Interfaces (APIs) are considered critical tools for bridging frontend applications with backend services. APIs are the basic conduit through which frontend applications communicate with server-side resources, enabling the seamless exchange of data and functionality that underpins today's dynamic web experiences.

Over the years, the use of APIs in frontend development has developed further, integrating a large number of advantages into the mix.

APIs provide a standardized way for applications to interact with external services, irrespective of the underlying implementation details. This abstraction simplifies the development process, allowing developers to focus on creating engaging user interfaces without worrying about the problems relating to server-side operations.

APIs are also known for their ability to facilitate modularity in application design. They bridge a range of specific functionalities into distinct APIs, allowing developers to build applications that are relatively easier to maintain and scale. This modularity also promotes code reuse, as the same API can be leveraged across different parts of an application or even across different applications.

APIs also make room for real-time data access and manipulation. This is crucial for creating interactive and responsive web applications. A prime example of this is how Meta intends to use APIs for real-time academic information update via the Researcher API. Through APIs, frontend applications can retrieve, update, and display data from backend services in real-time, providing users with a dynamic and engaging user experience[1].

However, despite these benefits, integrating APIs into frontend applications is not without its challenges. While APIs offer great potential, they also introduce new issues and potential pitfalls that developers must work through.

These challenges, which will be the focus of this paper, range from handling asynchronous operations and managing state, to securing sensitive data and ensuring application performance.

## 2. Literature Review

Various studies and reports have highlighted the challenges of API integration in frontend development, emphasizing issues such as the ones discussed here.

IBM researchers have proposed several solutions including standardizing data formats, improving documentation, implementing robust security measures, modernizing legacy systems, ensuring scalability, managing API versions, and implementing effective error handling.

## 3. Current Landscape

API integration is at the heart of modern software development. It gives room for applications to communicate seamlessly and share data. However, this process comes with its own set of challenges.

### 3.1 Incompatible Data Formats

One of the significant hurdles encountered in API integration pertains to managing incompatible data formats. APIs often adopt diverse data representation formats, including JSON, XML, and CSV.

For instance, while JSON has gained popularity due to its simplicity, XML remains prevalent in certain contexts. This diversity can lead to compatibility issues, necessitating additional effort to convert data between different formats.

For example, in cases where an application receives data in XML format from one API endpoint it may need to process it in JSON format for compatibility with another system. This conversion process adds challenges and may introduce errors if not handled properly. The conversion process may also open the API and in turn the website or application to security risks at the backend[2].

The frontend developer is primarily responsible for this, to collaborate with the backend developer to integrate this functionality and API properly.

### 3.2 Poor Documentation

Another obstacle that frequently impedes API integration is the prevalence of poor documentation. Effective utilization of APIs heavily relies on clear and up-to-date documentation.

However, not all APIs come with comprehensive documentation, which can complicate developers' understanding and lead to increased development time and errors[3].

For instance, a developer trying to integrate a new API into an application may struggle to decipher its functionality due to poorly documented endpoints and parameters. This lack of clarity can significantly hinder the integration process and delay project timelines.

### 3.3 API Security

Security becomes a top priority in API integration as data flows between connected systems increase. Strong security measures like encryption, authentication, and authorization are crucial to protect against potential threats[4].

### 3.4 Legacy Systems or APIs

Legacy systems without exposed endpoints pose another formidable challenge in API integration. These systems, not initially designed for seamless integration with modern applications, may lack the necessary interfaces to connect with external APIs.

Retrofitting the legacy system or considering a complete overhaul becomes necessary to enable API integration[5]. For instance, imagine a company with an outdated inventory management system that lacks API support.

Integrating this system with a new e-commerce platform requires significant effort to develop custom APIs or migrate to a more modern system with built-in API capabilities.

### 3.5 Scalability if Applications of Websites

Scalability is yet another important consideration as applications and websites expand. As a result, they are seeing increased user traffic.

The surge in API calls can strain the system. This leads to performance issues and potential downtime.

Ensuring scalability is vital right from the start. Take, for instance, a social media platform that's rapidly gaining users. It needs to design its API infrastructure to handle more requests without sacrificing performance or reliability.

Rate limiting, a method used by API providers to manage the number of requests in a given timeframe, brings its own difficulties. While it's important for preventing misuse and ensuring equitable usage, surpassing the rate limit can lead to unsuccessful API calls, diminishing the user experience.

### 3.6 API Versioning

API versioning introduces another layer of difficulty to the integration process. This strategy allows providers to introduce changes to the API without breaking existing integrations[6].

However, managing different versions requires diligence on the part of developers. They must ensure compatibility with the API version in use and be prepared to update their application with each new release.

### 3.7 Error Handling

Proper error handling emerges as a crucial factor for seamless API integration. APIs can return various error responses, ranging from server errors to validation issues.

Currently, the most prominent challenges that the frontend development team has to face in terms of API integration include:

- Performance issues after integration (55%)
- Scalability (47%)
- Securing third party partnerships (43%)
- Documentation-related issues (38%)[7].

## 4. Proposed Solution

Addressing the challenges of API integration requires a rather broad approach. Here are some proposed solutions to the challenges discussed in the previous section:

### 4.1 Standardizing Data Formats and Protocols

To standardize data formats across APIs, JSON (JavaScript Object Notation) stands out as a widely accepted standard. JSON's simplicity and compatibility with JavaScript, the primary language in frontend development, make it an ideal choice. Let's see an example of how JSON simplifies data exchange:

```
// Example JSON data representing a user profile
const userProfile = {
   "id": 12345,
   "username": "john_doe",
   "email": "john.doe@example.com",
   "age": 30,
   "city": "New York" };
// Convert JSON data to string for transmission
const jsonData = JSON.stringify(userProfile);
// Transmit jsonData to the API endpoint
// Here, the API expects data in JSON format
```

In this example, we have a user profile represented as JSON data. The *JSON.stringify()* function converts the JavaScript object *userProfile* into a JSON string for transmission to the API endpoint. Using JSON ensures that data is formatted consistently, making it easier to handle on both the frontend and backend.

### 4.2 Improving Documentation

API documentation is a key aspect for ensuring ease of use. It is also importtant for better integration of the APIs.

Swagger and Postman are tools that automate the creation and upkeep of API documentation, guaranteeing precision and ease of access for developers. Let's take a look at how Swagger simplifies the process of documenting APIs.

```
# Swagger YAML example for documenting an API endpoint paths:
 /users/{userId}:
  get:
   summary: Get user by ID
   parameters:
    - in: path
     name: userId
     required: true
     description: ID of the user to retrieve
     schema:
      type: integer
      format: int64
   responses:
    '200':
     description: Successful operation content:
      application/json:
       schema:
        $ref: '#/components/schemas/User'
```

In this instance, the YAML syntax offers a structured approach to documenting API endpoints, covering parameters, responses, and data schemas. This enables developers to grasp how to engage with the API and manage responses according to the documented guidelines.

### 4.3 Modernizing Legacy Systems

Modernizing legacy systems involves exposing APIs or utilizing middleware to integrate them with modern frontend applications.

Via RESTful APIs, the functionality of a legacy system can be exposed. Here is a code snippet of how this can be achieved.

```
// Example Java code for exposing legacy system functionality as RESTful
APIs
@RestController
@RequestMapping("/legacy")
public class LegacyController {
   @Autowired
   private LegacyService legacyService;
   @GetMapping("/data")
   public ResponseEntity<List<Data>> getAllData() {
      List<Data> data = legacyService.getAllData();
      return ResponseEntity.ok().body(data);     }
   // Other API endpoints for CRUD operations
}
```

In this Java Spring Boot example, the *Legacy Controller* exposes endpoints to interact with legacy system data.

### 4.4 Ensuring Scalability

Scalability is critical for handling increased loads as applications expand. Implementing caching mechanisms, deploying load balancers, and optimizing API performance are essential strategies.

Caching can be used to improve scalability as follows:

```
// Example of client-side caching in a frontend application
const cache = {};

const fetchData = async (url) => {
   if (cache[url]) {
      return cache[url];
   } else {
      const response = await fetch(url);
      const data = await response.json();
      cache[url] = data;
      return data;
   }
};
```

### 4.5. Handling Rate Limiting

Effectively managing rate limiting is crucial for frontend applications to operate within API constraints and optimize resource utilization. Let's explore some strategies and their implementation:

**4.5.1. Retry logic implementation:** Retry logic allows applications to automatically retry failed API requests after a brief delay.

Here's a simple example of implementing retry logic in Java Script:

```
const MAX_RETRIES = 3;
let retries = 0;

const fetchDataWithRetry = async (url) => {
   try {
      const response = await fetch(url);
      const data = await response.json();
      return data;
   } catch (error) {
      if (retries < MAX_RETRIES) {
         retries++;
         // Exponential backoff: increase delay with each retry
          const delay = Math.pow(2, retries) * 1000; // exponential
backoff in milliseconds
         setTimeout(() => fetchDataWithRetry(url), delay);
      } else {
         throw new Error('Max retries reached');
      }
   }
};
```

In this scenario, if the application's request to an API fails, it tries again up to a set number of times, which is defined as *MAX_RETRIES*. The time gap between each retry increases exponentially with every attempt. This approach helps prevent the server from getting overloaded with too many requests happening simultaneously.

### 4.6 Managing API Versions

Managing API versions ensures long-term compatibility and interoperability with evolving APIs. Let's explore some techniques and their implementation:

**4.6.1. URL versioning:** URL versioning involves including the API version in the request URL. Here's an example of URL versioning:

```
const API_URL = 'https://api.example.com/v1';

const fetchData = async () => {
   try {
      const response = await fetch(`${API_URL}/data`);
      const data = await response.json();
      return data;
   } catch (error) {
      console.error('An error occurred:', error.message);
   }
};
```

In this example, the API version (**v1**) is included in the URL (*https://api.example.com/v1/data*). This allows developers to introduce changes to the API while maintaining backward compatibility for existing clients.

**4.6.2. Header versioning:** Header versioning involves specifying the API version in the request headers. Here's how you can implement header versioning:

```
const headers = {
   'X-API-Version': '2.0'
};
const fetchData = async () => {
   try {
      const response = await fetch('https://api.example.com/data',
{ headers });
      const data = await response.json();
      return data;
   } catch (error) {
      console.error('An error occurred:', error.message);
   }
};
```

### 4.8. Implementing effective error handling

Effective error handling enhances application reliability and user satisfaction. Let's explore some best practices and their implementation:

**4.8.1. Retry mechanisms for transient errors:** Using retry mechanisms can help reduce transient errors resulting from temporary network issues or server downtime. Here's an example of how retry logic can be implemented for transient errors:

```
const MAX_RETRIES = 3;
let retries = 0;

const fetchDataWithRetry = async () => {
   try {
      const response = await fetch('https://api.example.com/data');
      const data = await response.json();
      return data;
   } catch (error) {
         if (error instanceof TransientError && retries < MAX_
RETRIES) {
         retries++;
            setTimeout(fetchDataWithRetry, 1000); // Retry after 1
second
      } else {
         console.error('An error occurred:', error.message);
      }
   }
};
```

In this example, if the API request fails due to a transient error, the application retries the request up to a maximum number of times (*MAX_RETRIES*). This helps mitigate temporary issues and improves overall system resilience.

### 4.8.2. Providing informative user feedback

Providing informative user feedback helps users understand the nature of the error and how to proceed. For this;

```
const handleApiError = (error) => {
   if (error instanceof RateLimitExceededError) {
      alert('Rate limit exceeded. Please try again later.');
   } else if (error instanceof AuthenticationError) {
      alert('Authentication failed. Please log in again.');
   } else {
      alert('An unexpected error occurred. Please try again later.');
   }
};
```

In this example, different types of errors (e.g., rate limit exceeded, authentication failure) trigger different alert messages, providing users with actionable information. This helps manage user expectations and enhances the overall user experience.

## 4. Academic Review of Perceived Challenges

**Table 1:** Table of studied literature regarding challenges.

| Name | Title | Challenge Discussed |
|---|---|---|
| K. Kim | APIs for real-time distributed object programming | Real-time distributed object programming with APIs |
| Q. W. & L. G. Ronghua Sun | Research Towards Key Issues of API Security | Key security issues in API development |
| S. M. S. A. S. Michael Meng | Optimizing API Documentation: Some Guidelines and Effects | Guidelines and effects of optimized API documentation |
| V. Bourne | 2021 State of Enterprise APIs | Trends and challenges in enterprise API usage |
| S. Salmone | Major Integration Challenges of Today and How to Overcome Them | Overcoming integration challenges in modern environments |
| S. Sohan, C. Anslow and F. Maurer | A Case Study of Web API Evolution | Evolutionary patterns and challenges in Web API development |
| | | |

## 5. Case Study

Axis Bank, a prominent private sector bank in India, initiated a digital transformation initiative to enhance customer experiences and streamline banking services. However, they encountered various challenges related to API integration, notably poor documentation, legacy systems, and scalability concerns.

The incomplete and unclear API documentation posed a considerable barrier as well. It impacted the efficiency of integration processes.

Moreover, the presence of legacy systems, ill-equipped for modern API interactions, complicated integration efforts, necessitating a gradual modernization approach. As customer demands increased, Axis Bank faced the critical need for a scalable solution to accommodate the burgeoning API traffic.

To address these challenges, Axis Bank utilized industry-standard JSON for API responses and requests, the bank standardized data formats and protocols.

Complementing this standardization effort, Axis Bank also invested significantly in enhancing API documentation, providing clear usage instructions and best practices to facilitate seamless integration processes.

Throughout the process, the bank undertook a phased approach to modernize its legacy systems, transitioning towards microservices architecture to expose specific functionalities via APIs.

Axis Bank used API gateways such as IBM API Connect to manage API traffic and enforce security protocols. The entire digital transformation was based on several programming languages including Java, Node.js, and Python, while using cloud platforms like Amazon Web Services and Microsoft Azure to host APIs, ensuring scalability and availability.

Axis Bank resultantly enhanced its digital banking services, improving customer experiences significantly in India. Furthermore, streamlined integration processes facilitated faster deployment of new features and services, reducing time-to-market and increasing operational efficiency[8].

## 6. Conclusion

The challenges of API integration in frontend development has no one-size-fits all approach. Instead, it requires a blend of different approaches that address compatibility, documentation, security, legacy systems, scalability, versioning, error handling, and performance issues.

Despite the issues and potential pitfalls associated with API integration, frontend developers can use the solutions detailed above to improve the integration process for the present as well as the future.

## 7. References

1. K Kim. APIs for real-time distributed object programming. Computer, 2000; 33: 72-80.
2. Ronghua Sun, Qianxun Wang, Liang Guo. Research towards key issues of API security. In: China Cyber Security Annual Conference, 2021;6: 4.
3. Michael Meng, Stephanie M. Steinhardt, Andreas Schubert. Optimizing API documentation: Some guidelines and effects. Proceedings of the 38th ACM International Conference on Design of Communication, 2020; 24: 1-11.
4. https://rapidapi.com/uploads/WP_2021_Enterprise_So_AP_Is_Report_f7fa0f3feb.pdf
5. https://www.ibm.com/products/cloud-pak-for-integration
6. S Sohan, C Anslow, F Maurer. A case study of web API evolution. IEEE, 2015; 27: 6.
7. https://dl.acm.org/doi/abs/10.1145/3470133
8. https://www.ibm.com/case-studies/axis-bank