



Unified

Presented by [Unified.to](https://unified.to)

2024 State of SaaS APIs

Table of Contents

Introduction

API Design

API Specifications
API Documentation
API URLs
Data Centers

Authentication & Authorization

Authorization
OAuth
OAuth2
API Key Format
API Key Location
OpenID Authentication

Webhooks

Pagination

Conclusion

Glossary

Unified.to

Introduction

With more than 30,000 applications, the Software as a Service (SaaS) industry has become increasingly dependent on sharing user data between applications through APIs. Just as these SaaS applications have grown in complexity, the APIs have become more specialized. For example, payroll functionality comes from HRIS APIs, and sales intelligence solutions rely on CRM APIs. We no longer build stand-alone products that are entirely “home-grown” — instead, we build products on top of platforms, where APIs are the connective tissue.

The report provides an in-depth analysis of the current landscape of Software as a Service (SaaS) APIs across various industries. It explores key aspects of authentication and authorization, API design, and the ways in which information is requested and retrieved from APIs, shedding light on prevalent trends and practices within the API ecosystem.

These insights, based on over 20 years of experience developing API integration software, can help you make informed decisions about implementing or using APIs. We’ve used these insights in developing our platform, which integrates APIs from 145 providers across several SaaS verticals into a single, unified API. We believe you’ll find this document useful as you review SaaS application APIs.

API Design

API Specifications

How are APIs documented, defined and shared?

The most common API specification formats:

1) OpenAPI and Swagger

These are specifications for describing RESTful APIs. It provides a detailed description of an API's endpoints, parameters, responses, and other details, and can be written in JSON or YAML format.

2) Postman Collections

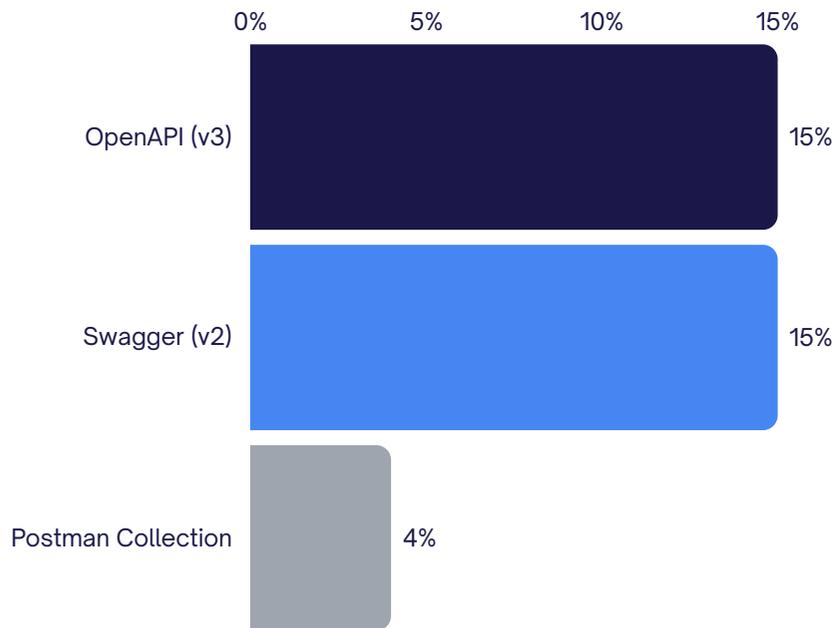
Designed for use with Postman, Postman Collection is an API specification that allows developers to define APIs, share and execute API requests, create and model API workflows, and document and test APIs.

3) RAML (RESTful API Modeling Language)

This YAML-based API specification focuses on top-down API modeling to describe APIs in a clear, concise way. Its structure makes it useful for long-term planning for APIs.

4) API Blueprint

This is a high-level documentation authoring language for describing web APIs that uses Markdown. As a result, it is easily readable by both humans and machines.



Supported API Specifications

Having an API specification makes it easier for your API consumers to integrate into your API. SDKs, documentation, and tests can be easily generated from a well-defined API specification.

OpenAPI and Swagger are the most supported API specification formats.

API Documentation

There are two ways to create API documentation:

1) Manually: Initially, API documentation was often manually created. This involves writing detailed guides, example requests and responses, and instructions for integrating with the API. Although this approach allows for a high degree of customization, it can be time-consuming to maintain and update, and the process is tedious.

2) Using automated tools: Tools like Swagger, Apiary, and Postman can automatically generate documentation from the API's codebase or from an API description file. The documentation generated by these tools is interactive, allowing developers to make API calls directly from the documentation to see real responses.

Some large language models are proficient at generating documentation from codebases, and it should be expected that AI will soon be integrated into automated API documentation tools.

While there isn't much data on the topic, it has been our experience that 50% of the API documentation we have encountered is either out of date or inaccurately describes the current API.

URLs

A URL, short for Uniform Resource Locator, which most users commonly understand as the location of a web page. In the context of APIs, URLs uniquely identify resources on the server, where “resource” could be anything the API provides access to, such as a user, a photo, a document, or other data. Each resource has a specific corresponding URL that clients can use to access it.

URL Types

Static Base URLs

A static base URL is a fixed URL that serves as the root address for accessing the API. It does not change across different customers or users of the API.

Custom Domain/Subdomain-based URLs

Custom domain or subdomain-based URLs allow the base URL of the API to be customized for individual customers or users, often using their domain name.

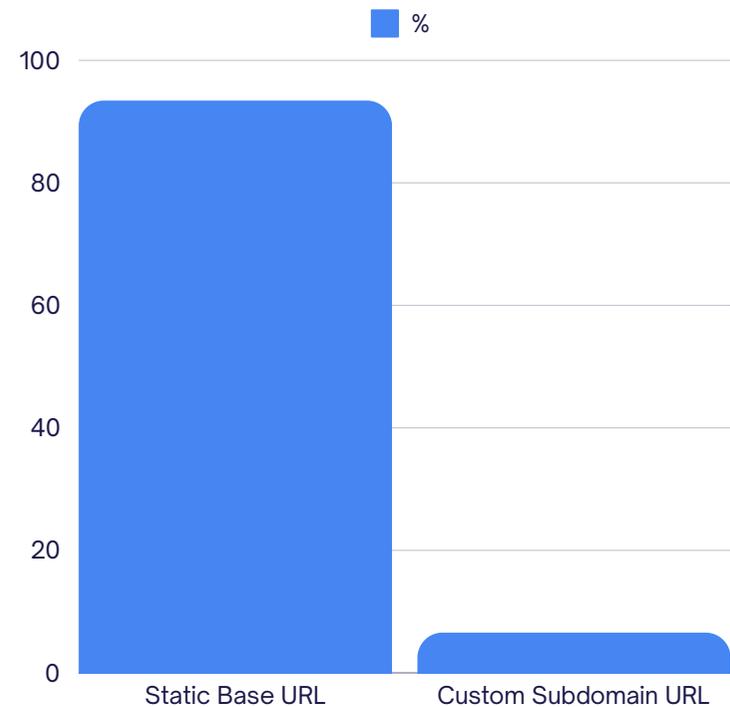
Versioned URLs

As a service evolves, so will the functionality and data it provides. This often leads to changes to the API, which will need to provide different URLs for new or updated functionality.

URL Usage

Static base URL at 93% is the most popular and widely used API URL.

This indicates most APIs utilize a consistent and unchanging base URL for accessing their resources. Those URLs that are customer or tenant specific potentially offer more security, but are harder to implement as an application needs to ask the end-user to retrieve it.

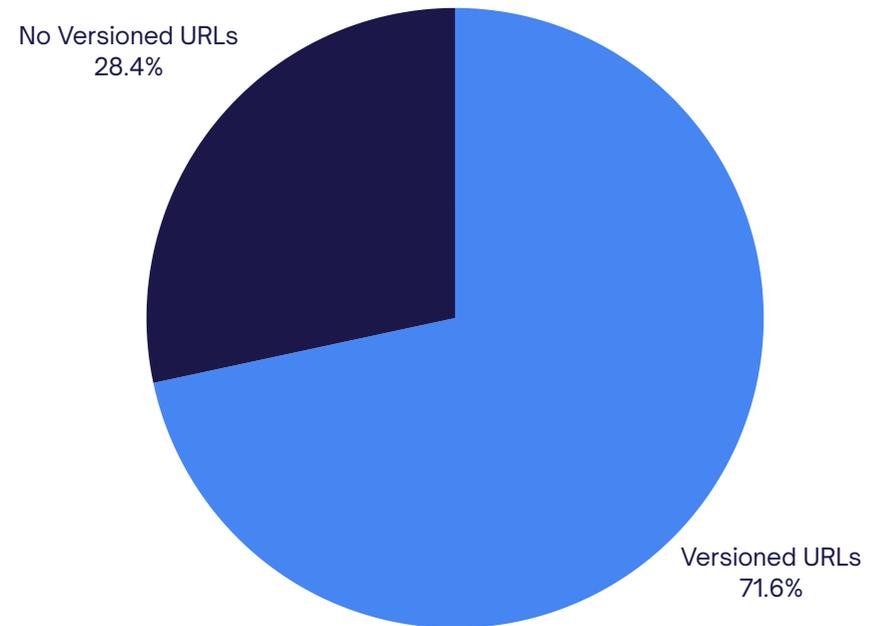


Versioned URLs

72% of APIs have versioned URLs

Versioning in a URL is key to advanced API design, ensuring backward compatibility, clear communication, and secure, targeted updates without disrupting current users. It facilitates parallel development and planned upgrades, allowing for a clear deprecation path and client-controlled adaptation.

This strategy enhances flexibility for future improvements while maintaining a stable and robust API environment, accommodating experimentation and feedback effectively.



Data Centers

A data center is a dedicated facility used to house computer systems and associated components, such as telecommunications and storage systems. It provides critical services including data storage, backup and recovery, data management, and networking. Data centers are essential for ensuring the continuous operation of application and providing connectivity to end-users. For the context of APIs, data centers can be categorized based on their geographical location or purpose.

Types of Data Centers

EU Data Centers: Located within the European Union, these data centers comply with EU regulations on data privacy and security, such as GDPR, ensuring that data handling practices meet stringent EU standards.

US Data Centers: Situated in the United States, these facilities are subject to U.S. laws and regulations, providing services primarily to American companies and users.

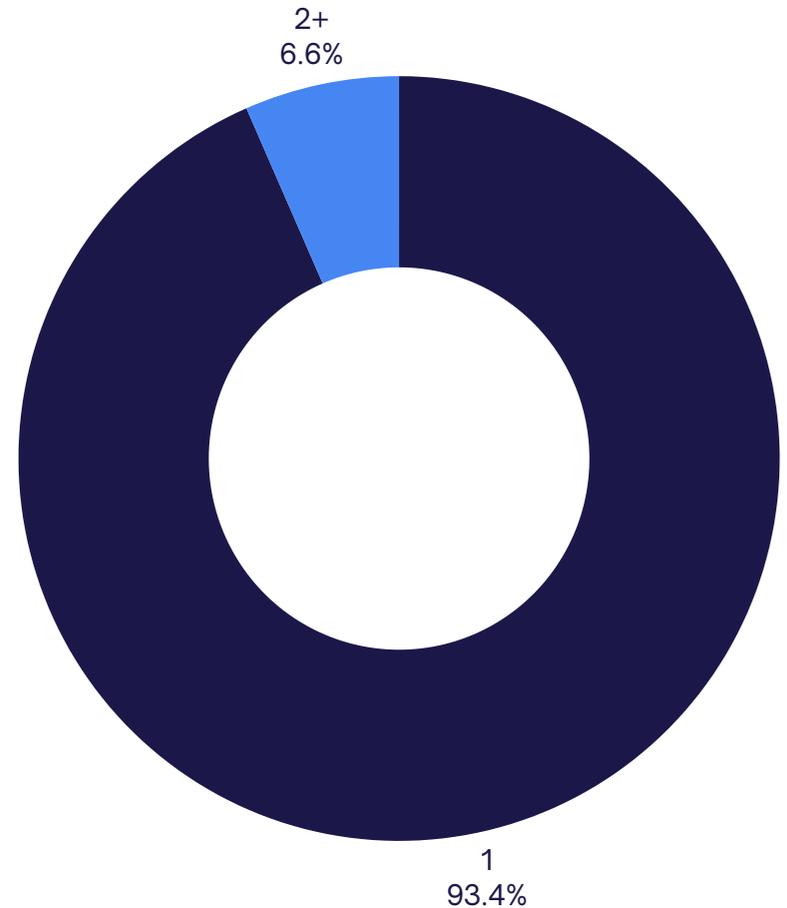
Sandbox Data Centers: These are testing environments where developers can experiment with APIs without affecting production data. Sandbox data centers simulate real-world operations, allowing for the safe testing of new features, bug fixes, and integrations.

Production Data Centers: These are live environments where APIs are deployed for actual use by clients and customers. Production data centers support real transactions and data processing, ensuring the availability, performance, and security of live applications and services.

Data Centers

How many data centers do APIs have?

The majority of APIs utilize only one API URL representing a single purpose or geographical location.



Authorization and Authentication

Basic Authentication (Username and Password)

The simplest scheme for securing an API is basic authentication, where the client authenticates itself with the server by including two text strings with every request:

- A username, which is who the client claims to be
- The corresponding password, which is a shared secret that *proves* that the client is who they claim to be

As part of the HTTP specification, basic authentication was meant as a simple way to grant website access to individual users. As a result, it can be used to identify the user making a request, but not the application they are using. Because this scheme identifies only users and not applications, it is a poor choice for controlling API access.

Basic authentication is the easiest authentication method to implement, but also the easiest to compromise. The username and password are sent as plain text as part of the request header, making it unsuitable for securing an API. A malicious party can easily get these credentials, allowing them to impersonate an approved client, or insert themselves between the client and the server to perform a “meddler-in-the-middle” attack.

API Key Authentication

API key authentication was designed to work around the limitations of basic authentication for API access. In this scheme, the client identifies itself to the server by providing a single text string — an *API key* — to identify the application attempting to access the API.

In a sense, API key authentication inverts the approach that basic authentication uses: instead of identifying the *user* accessing the API, it identifies the *application*. This approach simplifies the process of granting API access to a program or service and avoids the credentialing problems that arise with changes in staff. It also makes it possible for an API provider to track client usage and charge them accordingly.

This is a simple authentication method, which is why it is the most prevalent, accounting for over half (52%) of all authorizations.

The API key is typically stored either in the application's source code (which creates security risks) or in a configuration file. As a result, most API keys are accessible only by administrators.

API Key + Secret Authentication

This is a more secure variant of API key authentication, where the application provides not just an API key, but also a *secret* — a text string known only to the client and the API. The API key identifies the application, and the secret is used as proof that the calling application is what it claims to be.

Unlike the password in the basic authentication scheme, the secret is not directly included in the request. Instead, the calling application uses the secret in combination with other information from the request to generate a cryptographic signature that is then included in the request. The API server recalculates the signature using the same algorithm and the information it receives from the application to confirm that it is receiving requests from an approved client application.

OAuth2

OAuth2 is the industry standard method for authorization defined by the Internet Engineering Task Force (IETF). It enables a third-party application to obtain limited access to a user's resources hosted on a server, without exposing the user's credentials to the third-party application. It is the preferred method for allowing secure and controlled access to web APIs. In exchange for security, OAuth2 introduces complexity by defining the following roles and objects:

- **Resource Owner:** This is the user or application who owns the data or services — the *resources* — and wants to grant access to them to a third-party application.
- **Client:** This is the third-party application requesting access to the protected resources on behalf of the resource owner.
- **Authorization Server:** This is a server responsible for authenticating the resource owner and issuing permission in the form of *access tokens* (see below) to the third-party application (the Client) to access the resources after the Resource Owner grants authorization. It performs two important security tasks: verifying the identity of the Resource Owner and confirming that the Client is authorized to access the requested resources.
- **Resource Server:** This is the server hosting the resources that the Client wants to access; in other words, it houses the API. The Resource Server validates the access tokens presented by the Client, granting access to the requested resources if the tokens are valid.
- **Access Token:** An access token is a credential representing the authorization granted to the client by the resource owner. The client presents this token to the resource server to access the protected resources. Access tokens are short-lived and typically have limited scopes, which are permissions for access to specific resources and actions granted to the Client.

OAuth2 Con't

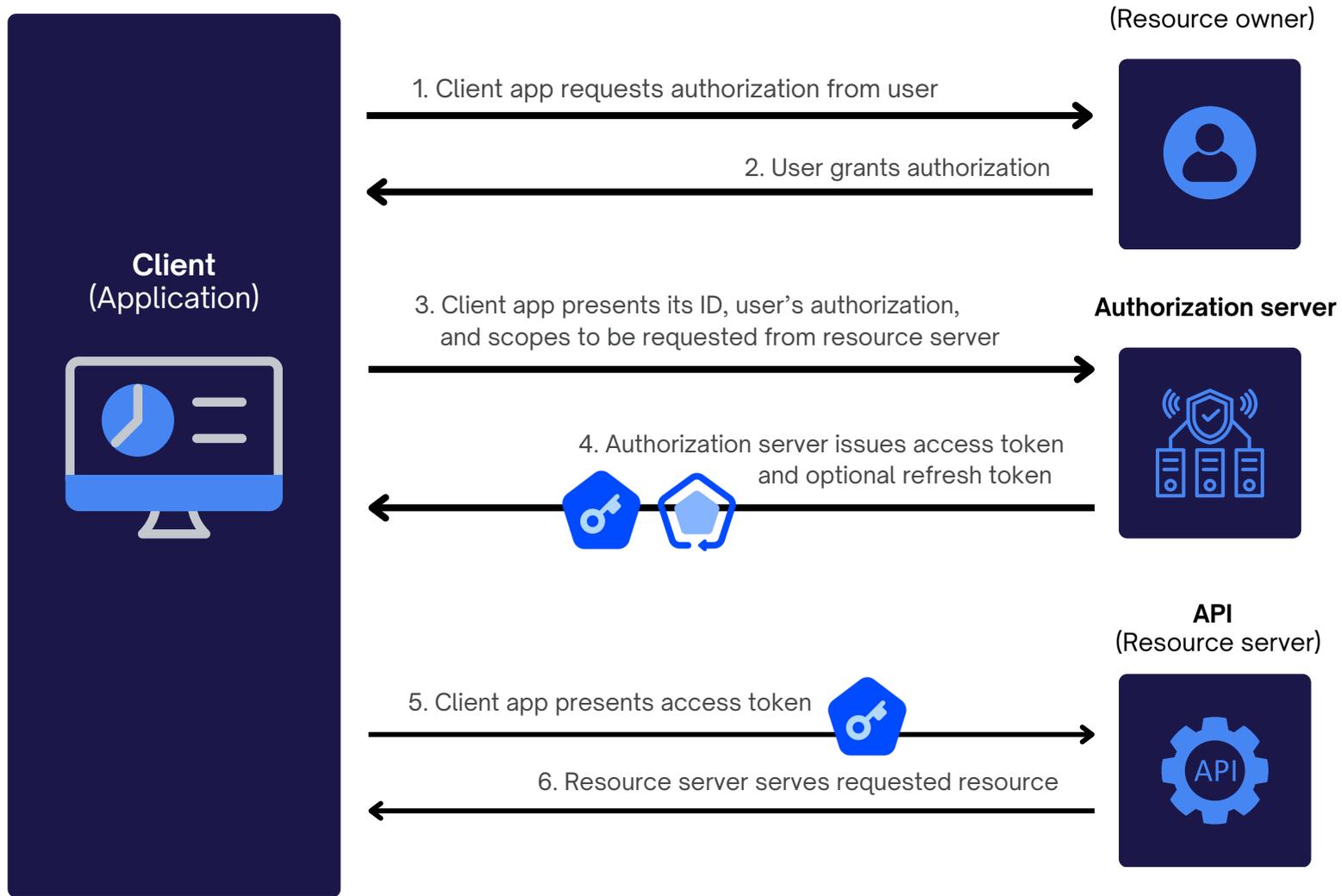
It should be noted that Access tokens are *bearer tokens*, which means that the token is *not* tied to a specific user or client; whoever holds the token can use it. If an unauthorized user were to get their hands on an access token, they could gain unauthorized access to the Resource Server and its APIs.

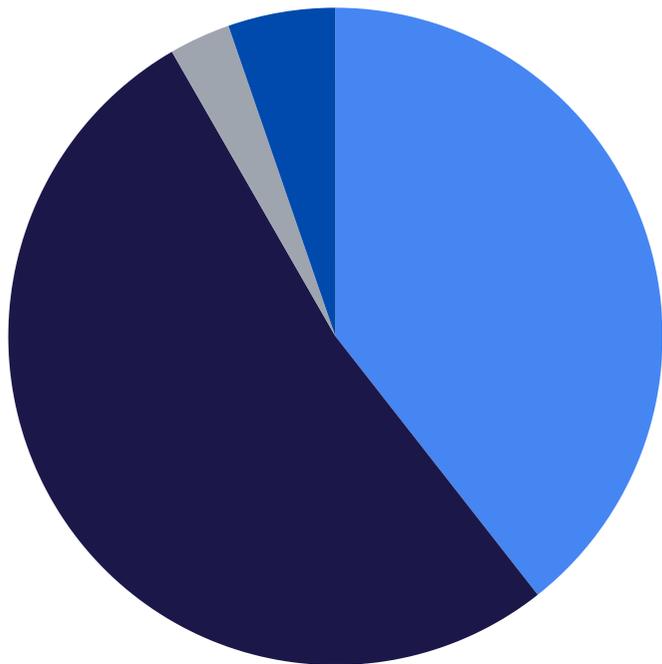
To mitigate the risks posed by compromised access tokens, an access token can be assigned a limited lifetime, after which it is invalid and will not be accepted by the Resource Server. This lifetime is specified in seconds and can be as short as one second and be as long as weeks.

When a Client's access token expires, it can no longer call the API on the Resource Server. The Client needs a new access token, which it can acquire in a couple of ways:

- 1) The Client can ask the User for authorization to call the API on the Resource Server. This requires the User to log in again, which is inconvenient, especially if the access tokens are short-lived
- 2) If the Authorization Server supports them, the Client can request a *refresh token* in addition to an access token. When the access token expires, the Client can send the refresh token to the Authorization Server, which will return a new access token in response. The Client can do this behind the scenes without having to ask the User for authorization.

The OAuth2 authorization flow is illustrated in the diagram below:





Authorization Methods

API keys are the most prevalent authorization method, accounting for over half (52%) of all authorizations.

OAuth2 is also widely adopted, representing 39% of authorization technology. OAuth2’s flexibility and added security features make it an increasingly popular choice among API providers. The following page explores the differences between each method.

Deciding between OAuth2 and API Key

From a customer-centric perspective, OAuth2 is undoubtedly the preferred authorization method. OAuth 2 is easier for end users as it displays a familiar authentication screen hosted by the source app. However, it's more difficult to configure for developers as it requires apps to initially be registered with the source app.

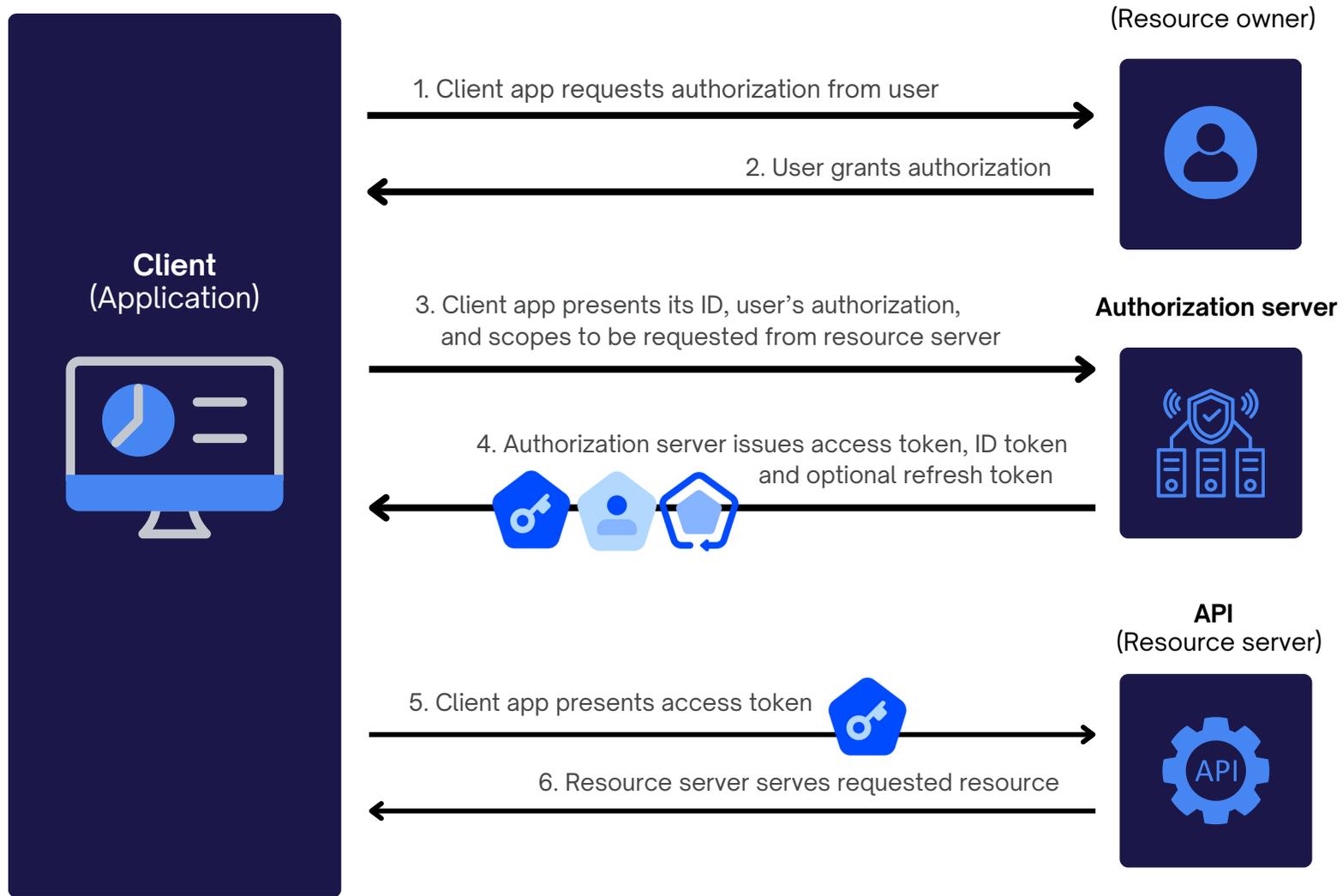
From a developer-perspective, API keys are the easiest authorization method as it requires zero configuration. However, end-users are required to locate their API keys in source applications, which are usually only accessible to admin users. This creates friction in the adoption of product integrations.

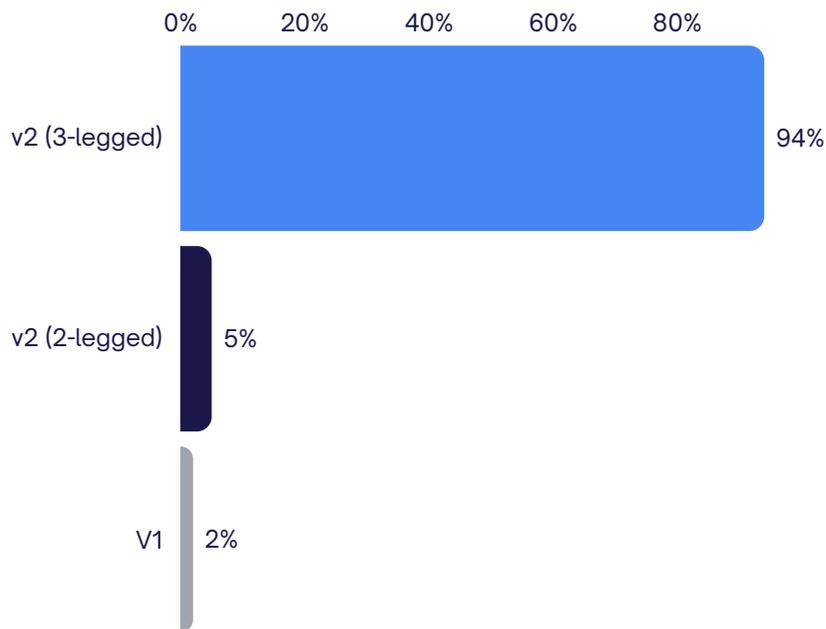
OAuth2 + OpenID Connect (OIDC)

OpenID Connect (OIDC) adds authentication to OAuth2, allowing a Client to confirm the identity of the end user based on authentication that the Authorization Server performs. It is defined by the OpenID Foundation, a non-profit organization that promotes, protects, and nurtures the OpenID community and technologies.

With OIDC, the Authorization Server also provides an ID Token, which contains information about the authenticated user, such as their unique identifier, email address, and optionally other user attributes and metadata.

The diagram below illustrates how OAuth2 and OIDC interact:





OAuth

This section reveals insights into the adoption of different types of OAuth versions. Authorization involves the verification of an individual or system's identity and credentials to gain authorized access to resources or perform specific actions within a third-party system.

OAuth2 using the 3-legged authorization flow is the most commonly used OAuth authentication method by developers, representing 94% of all OAuth authentications.

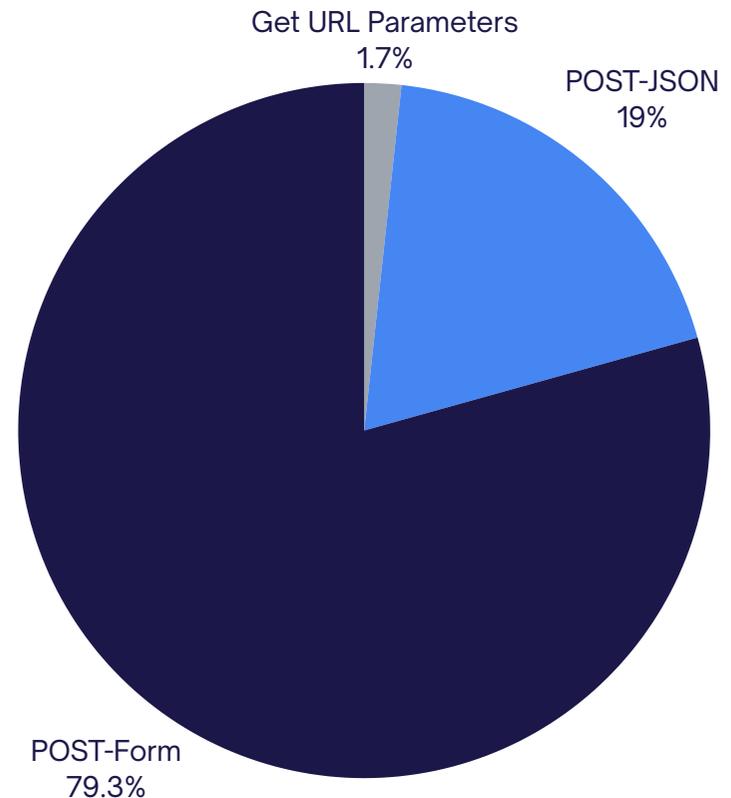
However, it poses a greater challenge for the requesting application, which must initially register their application with the third-party system before being able to authorize end-user accounts.

OAuth2 Token Endpoint

This section delves into the various methods used for authorizing OAuth2 requests and analyzes the distribution of these methods based on our dataset. **OAuth 2 requests using POST-Form authorization, where data is sent in form parameters, dominate the dataset with 79% usage.**

POST-JSON: In this method, authorization data is sent within the request body using JSON format. It accounts for 19% of OAuth2 authorizations. POST-JSON offers better security compared to GET parameters as it keeps sensitive data out of the URL.

URL parameters: This method involves sending authorization information within the URL's query parameters. It is used in 2% of OAuth2 authorizations. This method can be less secure due to potential exposure of sensitive data in URLs.



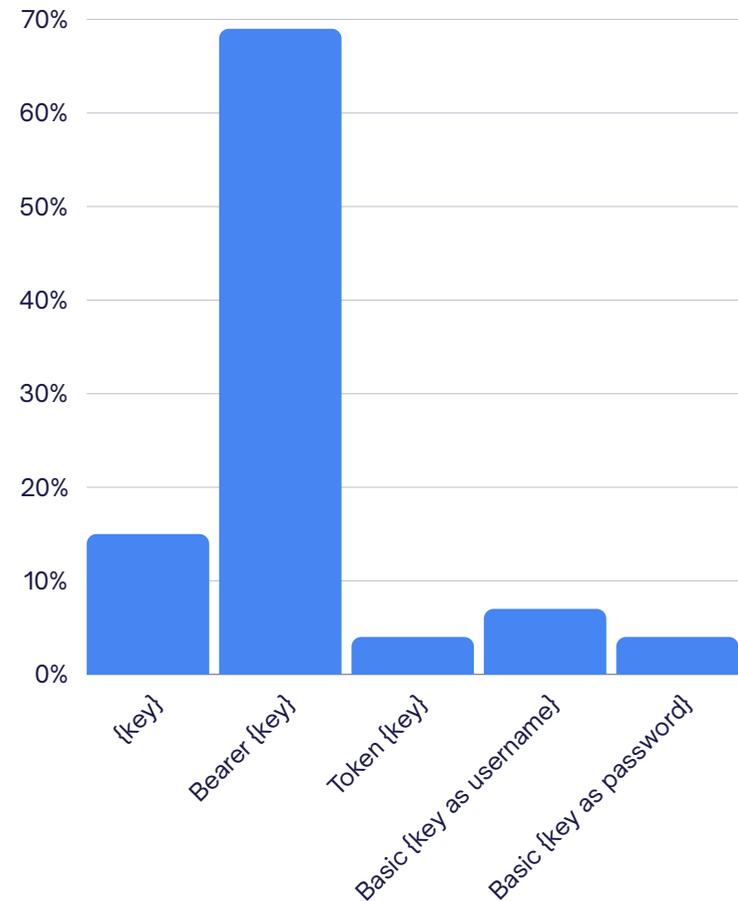
API Key/Token Formats

How are API keys/tokens sent to APIs?

The "Bearer" format is the most widely used API Key authorization format, accounting for the majority at 69%.

This is due to its use by OAuth2 APIs.

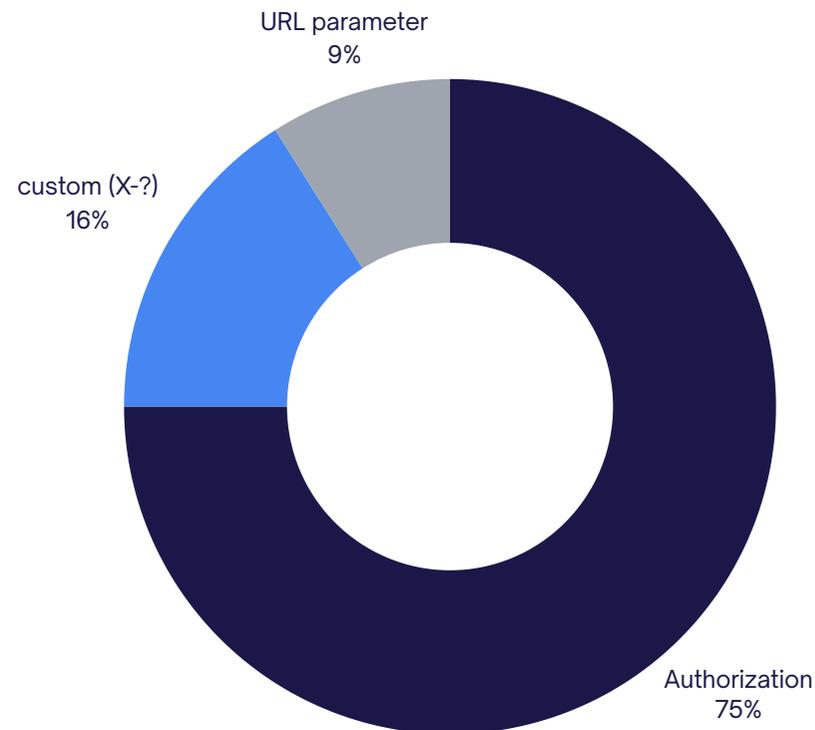
It is a common approach where the API token is included in the HTTP request headers, typically as an "Authorization" header, preceded by the word "Bearer."



API Key Location

Authorization header is the most popular location for an API key.

The Authorization header is the standard way to provide access credentials, but 25% of APIs in our dataset are not following this standard.





Sign in with Asana



Sign in with Discord



Sign in with Google



Sign in with HubSpot



Sign in with LinkedIn



Sign in with Microsoft

OpenID Authentication

OpenID Connect, a component of the OAuth2 specification, enables the requesting system to obtain the authorized user's identity information, such as their name and email.

APIs that incorporate OpenID simplify the implementation of a "Sign in with [platform name]" button or link.

Only 35% of OAuth2 APIs support OpenID Authentication.

Webhooks

Webhooks

Webhooks, also known as *web callbacks* or *push APIs*, are automated messages sent from an application or server to instantly notify another application or server when a specific event occurs. They send real-time data in response to events, allowing for immediate reactions to specific triggers. Webhooks offer a flexible and efficient method for building interconnected, responsive systems that can automatically exchange information and perform actions in real-time based on specific events.

Webhooks are recommended over traditional REST APIs in scenarios where timely, event-driven notifications are necessary. For example, webhooks can automate system notifications for status changes in an application, eliminating the need for manual polling. They simplify synchronization strategies by delivering up-to-date data to applications.

How Webhooks Work

1) Event trigger

A webhook is configured to listen for certain events in an application, such as a new user registration, an update to a candidate in an ATS, or the creation of a new deal in a CRM.

2) Notification

When the specified event occurs, the application sends a webhook payload, usually in JSON or XML format, to a URL configured to receive the messages. This URL is an endpoint on a server prepared to handle the incoming data.

3) Action

The receiving server processes the incoming webhook payload and performs a predefined action, such as updating a database, sending an email, or triggering another workflow.

Handling Webhook Failures

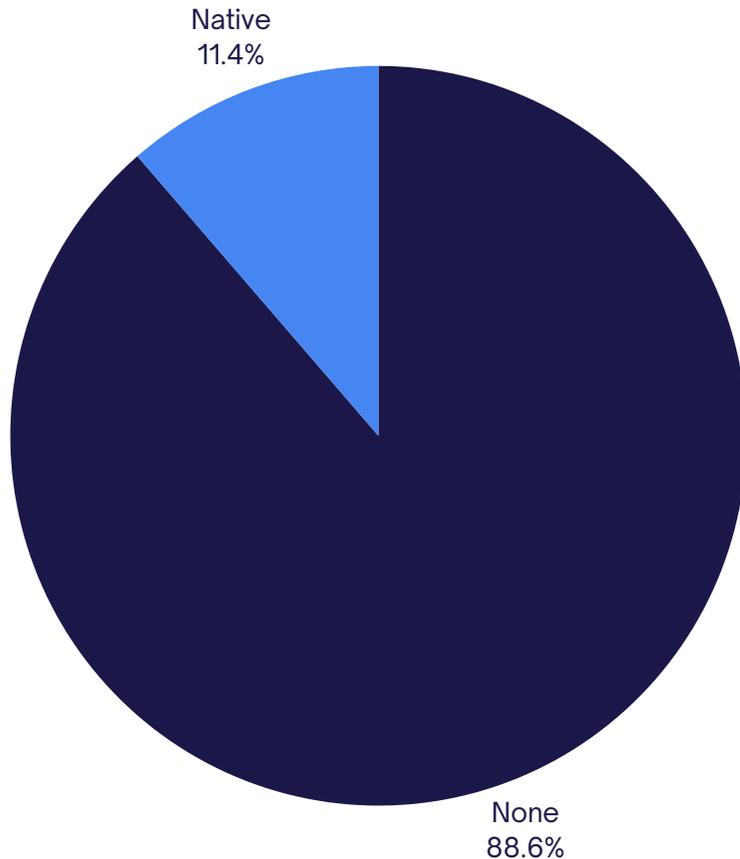
Retry mechanisms: Many services implement automatic retry policies for their webhooks. If a webhook fails to reach its destination because the receiving server is down or unreachable, the sending server might retry the request several times over a predetermined period. Some services use exponential backoff in their retry strategies, gradually increasing the interval between retries in order to reduce the load on the receiving server when it comes back online and avoids creating a denial-of-service condition.

Alerts: The sending service might notify the sender (via email, dashboard alert, or other mechanism) that a webhook delivery has failed. This allows developers or administrators to investigate and resolve the issue on the receiving end.

Logging and monitoring: Webhook transactions, including failures, are often logged. These logs can be crucial for debugging and understanding what went wrong. Monitoring tools can alert teams to repeated failures, indicating a problem that needs attention.

Dead letter queue (DLQ): Some systems implement this mechanism where failed webhook calls are stored for later processing. Once the issue with the receiving server is resolved, the queued webhook messages can be retried manually or automatically.

Native Webhook Support



Only 11% of API providers have built-in support for native webhooks, which forces developers to implement their own polling strategies.

In the absence of native webhooks, API consumers are required to develop sophisticated strategies and establish infrastructure for scheduled data polling. This entails managing rate-limiting and addressing various error scenarios.

Pagination

Pagination

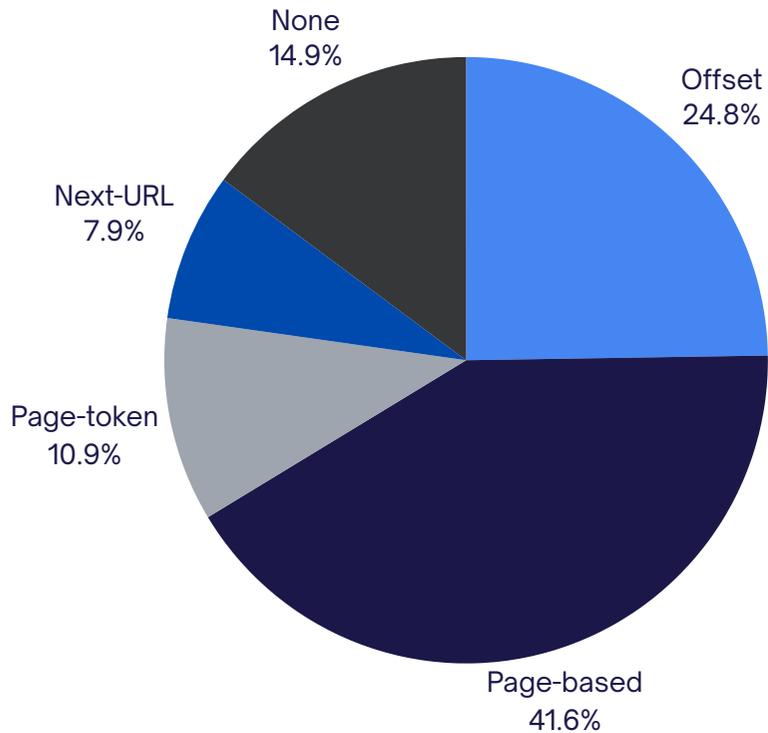
When an API returns a lot of data in response to a request, retrieving and processing all of this data at once can be inefficient, slow, and resource-intensive for both the server and the client. *Pagination* helps to alleviate these issues by allowing clients to request data in small sets or “pages,” making data retrieval more manageable and improving the overall performance and usability of the API. The common methods include:

Offset Pagination: Divides large datasets into "pages," letting clients skip a set number of records (offset) and define a maximum number of records to return (limit).

Page-based Pagination: Segments data into numbered pages for sequential navigation, enhancing user experience by abstracting data ranges into an intuitive sequence.

Page-token Pagination: Uses a unique token for data retrieval, ideal for constantly updating datasets, ensuring efficiency and reliability. Also known as token-based, sync-token, or cursor pagination.

Next-URL Pagination: Incorporates a direct link to the next set of results within the API response, streamlining the request process for clients.



Pagination

How do APIs structure paginated data?

Paging is the most common pagination method, which involves specifying a page number to access specific portions of pagination data.

PAGINATION METHOD	Pro	Con
OFFSET	<ul style="list-style-type: none"> • Precise control over the data range retrieved • Easy to implement and understand 	<ul style="list-style-type: none"> • Inefficient for large datasets since it requires scanning through previous pages. • Vulnerable to performance issues with increasing offset values
PAGE	<ul style="list-style-type: none"> • Simpler to use than offset, as it focuses on pages rather than specific offsets • Suitable for smaller datasets 	<ul style="list-style-type: none"> • May still suffer from performance issues with very large datasets • Not ideal for maintaining a consistent view if data changes between paginations
PAGE-TOKEN	<ul style="list-style-type: none"> • Efficient for large datasets, as it provides a stable reference point • Supports resuming pagination without losing data integrity 	<ul style="list-style-type: none"> • Requires more complex implementation • May require additional storage for page tokens
NEXT-URL	<ul style="list-style-type: none"> • Allows for straightforward navigation with well-defined URLs • Easy to implement 	<ul style="list-style-type: none"> • Depending on the API design, URLs may become unwieldy with deep nesting • May lack support for efficient random access to specific pages
NONE	<ul style="list-style-type: none"> • Simplest approach with no pagination logic to implement • Suitable for small, fixed-size datasets 	<ul style="list-style-type: none"> • Unsuitable for handling large datasets efficiently • May result in slow performance and increased bandwidth usage

Conclusion

Application development has become increasingly complex over recent years, not solely due to the growing number of APIs that developers need to interact with but also because of the increasing complexity of the APIs themselves.

Modern applications often rely on a diverse set of external services, from universal ones such as authentication/authorization and cloud storage to task- and industry-specific services such as HR and CRM. Each of these services comes with its own API, and each API features its own endpoints, data formats, authentication mechanisms, approaches to pagination, and other implementations. The sheer variety and complexity of integrating these disparate systems can significantly increase development time, elevate the risk of bugs, and complicate maintenance efforts. As APIs evolve, keeping up with changes in their specifications requires ongoing vigilance and effort from development teams.

API unification services can be a game-changer for developers. These platforms act as intermediaries between your application and all the external APIs it uses, providing a simplified, single interface for integration. By abstracting away the differences between individual APIs, these solutions can drastically reduce the complexity of dealing with multiple services, streamline the development process, and improve maintainability. They also often offer additional features like unified analytics, centralized error handling, and cross-API data normalization, further enhancing developer productivity and application reliability. As the landscape of web services continues to grow in both size and complexity, leveraging an API unification SaaS can be a strategic move to keep development efficient and focused.

List of APIs examined in this report: <https://unified.to/integrations>

Glossary

Access Token

A piece of digital information that authorizes the person or application possessing it to access specific data or functionality on a given system.

API

Application Programming Interface — a set of functions that allow one application to use the functionality of another application, just as a user interface allows a human user to use the functionality of an application.

API Blueprint

An authoring language based on the Markdown markup language for describing web APIs.

Authentication

The process where a person or system verifies that they are who they claim to be; essentially answers the question “Who are you?” Often used in combination with (and confused for) authorization. Sometimes shortened to *AuthN*.

Authorization

The process where a person or system is given the ability to access a resource or perform an action; essentially answers the question “What are you allowed to do?” Often used in combination with (and confused for) authentication. Sometimes shortened to *AuthZ*.

Glossary

Bearer Token

A piece of digital information that authorizes the person or application possessing it to access specific data or functionality on a given system.

Data Center

A physical location containing computer systems and their components that provide computation and data storage services.

Data Residency

The physical or geographical location of an individual or organization's data.

Data Sovereignty

The concept of a country or other jurisdiction's rights and control over data within its borders.

HTTP

Hypertext Transfer Protocol, which defines how resources are fetched and accessed over the internet, and primarily the World Wide Web.

ID Token

A piece of digital information that proves that a user has been authenticated.

OAuth2

Short for OAuth 2.0, an interaction standard that allows a website or application to access resources hosted by other web apps on behalf of a user.

Glossary

OpenAPI

A programming language-agnostic specification language for defining the structure and syntax of HTTP-based APIs.

OpenID Connect

A protocol built as a layer OAuth2 to authenticate users. Often shortened to OIDC.

Pagination

An approach used by APIs that allows clients to request data in small, easier-to-consume amounts instead of receiving the data as a giant set.

RAML

RESTful API Modeling Language, a YAML-based language for defining an API in a top-down fashion.

REST

REpresentational State Transfer, an architectural style for developing web services that is based on HTTP.

URL

Uniform Resource Locator, a reference to a resource that specifies its location on a computer network and a mechanism for retrieving it. Colloquially known as a “web address.

Webhook

An automated message sent from an application or server to instantly notify another application or server when a specific event occurs.

About Unified.to

Unified.to is an award-winning unified API development platform for B2B SaaS, covering 150+ industry-leading APIs across HR, Recruitment, Sales, Marketing, Authentication and more. With Unified.to, software companies integrate once to access their customers' hiring, selling, and workforce data from multiple systems to enable automation, insights, and better user experiences for the customers they serve. Software companies like Checkr, Cognism, Humi, and Sailes trust Unified.to to power their API integrations.

Unified.to is founded by Roy Pereira (CEO) and Alexey Adamsky (CTO), serial entrepreneurs and technologists with 20+ years of API integration experience.

Contact

hello@unified.to

www.unified.to