A guide to REST API authentication

Nataliia Fedchenko

API authentication secures a REST API by ensuring that only authenticated users can access its resources.

But how, exactly, does API authentication work? We'll answer this by breaking down popular authentication methods and how some well-known REST APIs implement API authentication.

What is REST API authentication?

It's the process of verifying a user's or application's identity before granting them access to your REST API's resources.

REST API authentication works differently than traditional web authentication, where users typically log in with a username and password and a session is created for them. Unlike the "remember me" feature on the login page of your favorite website, REST authentication, by default, doesn't hold on to session information. Each request acts independently, verifying itself without relying on past interactions. This stateless approach makes REST APIs scalable and robust.

Additionally, REST enables its clients to choose their preferred authentication method from a list of standardized options, like API keys, tokens, or OAuth. This flexibility encourages interoperability between different services.

REST API authentication vs authorization

It's also important to differentiate between authentication and authorization. Authentication confirms who you are, while authorization determines what you can do.

For example, API authentication might verify your application's identity, but authorization would then determine if your application has permission to modify data or access sensitive resources. Even after being authenticated, a user might not have access to all the API's resources or functionalities.

REST API authentication acts as a good first line of security for REST APIs. It can then be coupled with authorization to ensure that security constructs like the principle of least privilege (PoLP) are implemented correctly in your API.

Related: What is REST API integration?

Common REST API authentication methods

There are several methods for implementing authentication in the REST API. It's important to know the advantages and disadvantages of each so that you can choose the most suitable one based on the requirements, use cases, and constraints of your API.

Basic authentication

Basic authentication is a simple and fast method of HTTP authentication. To access the API endpoint, the user must send a username and password to the API provider in the authentication header of the request. The API provider checks the credentials and, in the case of success, grants access to the user.

Let's look at how you'd implement basic authentication in Python. This example shows how you can encode credentials with the base64 library and then add them to the Authorization header with the Basic prefix. You can then use this header to access the API endpoint with the requests.get() method:

```
import requests, base64
url = 'https://restapi.test.com/api/
credentials = 'user:password'
encoded = base64.b64encode(credentia
headers = {
    'Authorization': f'Basic {encode
}
response = requests.get(url, headers
print(response.status_code)
print(response.json())
```

Basic authentication is simple to implement and has a very low processing overhead on the server. However, the credentials sent as part of such a request travel unencrypted (possibly only encoded), which makes them vulnerable to interception. Also, since basic auth is quite simple and straightforward, you lack advanced control, such as granular access provisioning and revocation.

It would make sense to use basic auth when building internal APIs, for testing purposes, or when dealing with trusted clients in secure environments. However, if you're handling sensitive data in public-facing APIs or your use case requires fine-grained

access control, you should look at some of the more secure alternatives in this list.

Related: What you need to know about REST API rate limits

API key authentication

In API key authentication, the API provider assigns a unique key to each client accessing the API. The client needs to include their API key as part of the request to authenticate themselves. The API key can be included anywhere in the request, such as the header, body, or query parameters. It ultimately depends on the API's design and is communicated to the developers via the API documentation.

The following example shows how to authenticate with a REST API that uses API keys in Python using the requests library. It adds a string that contains the API key in the authorization header with the prefix api-key and uses this header when sending a request to the API.

```
import requests
url = 'https://restapi.test.com/api/
api_key='some_secret_key'
headers = {
    'Authorization': f'api-key {api_
}
response = requests.get(url, headers
print(response.status_code)
print(response.json())
```

API key authentication is particularly straightforward to implement. You can even assign dedicated sets of permissions to individual API keys, making it a popular choice for scenarios where a moderate level of security is sufficient. Another

benefit is that it allows you to implement API key revocation. If a key is compromised, the server can revoke it so that the client can request a new one.

However, this method is only moderately secure. If the API network is vulnerable, attackers can intercept the API key and use it for malicious purposes until it expires or is disabled manually. Therefore, avoid using this method when handling highly-sensitive data or where you need to track user activity for logging or auditing purposes.

Related: What is an API key?

Token-based authentication

Token-based authentication—also called bearer authentication—is a popular authentication method that uses an access token to verify a user's identity.

While API keys are used to identify clients—i.e. the site or the app making a call—, tokens are used to identify users because they contain additional information about the user. Unlike the other methods discussed so far, tokens can be dynamic, meaning you can generate tokens on the fly using a dedicated token-generation flow built into the API. This allows unique, short-lived tokens to be generated based on the immediate use case of the users.

Tokens can be customized to make them device- or user-specific. This means that you can also (to some extent) verify the authenticity of a token on your server before processing a request. This further enhances the security of your API.

Let's consider an example of authenticating with a REST API that requires a bearer token. To access the API endpoint, simply add a token with the prefix "Bearer" to the authorization header of the request:

```
import requests
url = 'https://restapi.test.com/api/
token='fhu78ej3-fh37-fy67-56ed-56ddg
headers = {
    'Authorization': f'Bearer {token
}
response = requests.get(url, headers
print(response.status_code)
print(response.json())
```

Token-based authentication is one of the most secure methods of authenticating REST APIs. They minimize interception risk since they can be encrypted and are usually short-lived, and they can offer granular access control.

However, tokens require careful design and infrastructure considerations. They are, in general, more complex to implement compared to other methods, and the process of generating tokens can add some processing overhead to the server.

Using token-based authentication makes the most sense when handling sensitive data in public APIs or when building complex, scalable API ecosystems. If simplicity and resource efficiency are your primary concerns, you are better off using a simpler method.

Real-life examples of REST API authentication

To better understand how the various API authentication methods work in practice, let's consider how some well-known products implement authentication for their REST APIs.

BambooHR

API to integrate with third-party applications.

BambooHR's REST API uses an API key for authentication. The API key is sent through basic HTTP authentication as a username, and the password can be any random string.

To gain access to its API endpoints, you need to add credentials in the format "{user}:{password}" in the authorization header with the Basic prefix. You then need to pass the API key as the user and a random string as the password.

```
import requests
url = 'https://api.bamboohr.com/api/
user = 'API_key'
password = 'some_random_string'
headers = {
    'Authorization': f'Basic {user}:
}
response = requests.get(url, headers
print(response.status_code)
print(response.json())
```

Greenhouse

Greenhouse is a popular applicant tracking system (ATS) that offers five different APIs.

One of them, Harvest—which allows you to manage open jobs, candidate data, and more—uses basic authentication. To gain access to the Harvest API, you need to pass the API key as the username and leave the password blank.

For example, if the authorization header value looks like "{user}"`, where the user is the API key, the authentication code can look as follows:

```
import requests
url = 'https://harvest.greenhouse.io
username = 'API_key'
headers = {
    'Authorization': f'Basic {user}:
}
response = requests.get(url, headers
print(response.status_code)
print(response.json())
```

Box

Box is a cloud platform that provides tools for content management, collaboration, digital signature, and other business operations. It ensures API security with token-based authentication. Depending on the API's functions in the app, the app uses different methods to obtain the token: OAuth 2.0, JWT, or the Client Credentials grant.

Here's how the authentication code can look:

```
import requests
url = 'https://api.box.com/2.0/users
token = 'some_secure_token'
headers = {
    'Authorization': f'Bearer {token
}
response = requests.get(url, headers
print(response.status_code)
print(response.json())
```

Related: A guide to testing API integrations

Best practices for API authentication

Following best practices can protect your API from hacking and unauthorized access. We'll consider some of the most important ones below.

Enforce strong passwords and implement password hashing

For basic authentication, it's important to use strong passwords because they're difficult for attackers to guess.

Use password validation to require users to create strong passwords by enforcing a robust password policy:

- **Minimum length:** Enforce a minimum password length of twelve characters, preferably longer.
- **Complexity:** Require a combination of uppercase and lowercase letters, numbers, and special characters.
- **Uniqueness:** Encourage unique passwords for each API account, eliminating the reuse of one password across multiple services.
- **Regular updates:** Mandate regular password changes, ideally every three to six months, to limit the window of vulnerability.

Moreover, storing passwords in plain text is an invitation to disaster. Implement password hashing algorithms like Argon2 or bcrypt before storing passwords in your database. These algorithms transform passwords into irreversible strings, making them unusable even if intercepted.

Implement token revocation for token-based authentication

Token revocation is the process of invalidating a token before it expires.

If a token is ever compromised, token revocation allows you to revoke the token to prevent attackers from gaining access to sensitive information. Additionally, when you change a user's access rights or permissions, revoking the token and issuing a new one ensures that the updated permissions are enforced.

In OAuth 2.0, the token-revocation mechanism is implemented using a token-revocation endpoint. To revoke a token, clients must send a request to this endpoint, indicating which token needs to be revoked. Other authentication protocols may have different ways of carrying out token revocation.

Limit session length

Session length is the time a user spends on a website in one session. In the context of APIs, you can limit session length using the token's expiration.

When setting up token expiration for an API, it's important to strike a balance between security and user experience. Setting too short an expiration may result in forcing clients to reauthenticate too frequently, while setting it too long may cause a security risk. Consider the sensitivity of the data and operations provided by your API when determining token expiration.

Implement access control lists

An access control list (ACL) is a set of rules that determine which users and systems are allowed to access certain API resources and which actions they're allowed to perform.

ACLs are used alongside common API authentication methods to verify if the authenticated user is allowed access to the target resource. This adds another layer of

security over your regular token or OpenID-based access control constructs. If you're building highly sensitive public-facing APIs that allow access to multiple types of data through a common interface, it's best to employ ACLs to strengthen your security.

Related: 7 API integration best practices

Build secure integrations at scale with Merge

If you're looking to integrate your product with the 3rd-party applications your clients and prospects use, the process of accommodating to each API's approach to authentication can quickly overwhelm your engineers.

You can bypass this altogether by simply building to Merge's Unified API.

By building to our unified API, you'll also be able to access hundreds of integrations across popular software categories—from HRIS to CRM to ATS—and receive the maintenance support and management tooling you need to provide reliable and high-performing integrations.

You can learn more about Merge by scheduling a demo with one of our integration experts.

Read more

A guide to working with the SharePoint API

Engineering

What are API access controls? Plus tips for...

Insights

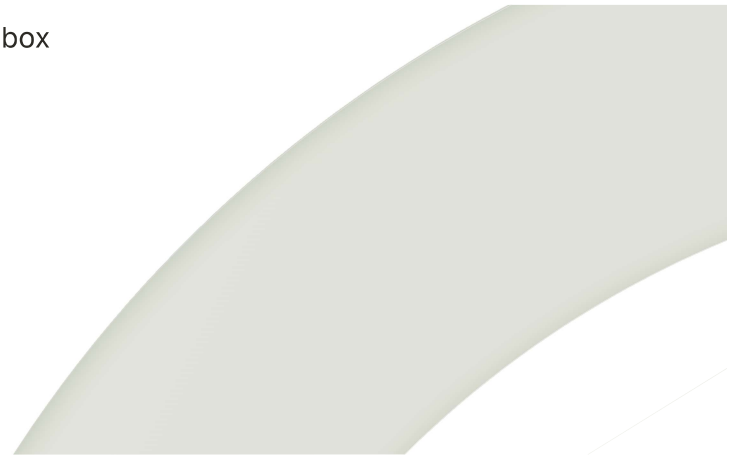
6 Model Context Protocol alternatives to...

AI

Subscribe to the Merge Blog

Get stories from Merge straight to your inbox

Subscribe



Integrations

- Accounting integrations
- ATS integrations
- CRM integrations
- File storage integrations
- HR integrations
- Ticketing integrations
- All integrations

Platform

- Why Merge
- How Merge works
- Localization
- Security
- Observability
- Common models
- Customization
- Developer tools
- Real-time data
- Platform architecture
- MCP

Personas

- Product
- Developers
- Go-to-market teams

Use cases

- Power AI features
- Auto-provision
- Knowledge base
- Financial analysis
- Candidate sourcing
- Project analysis
- Source leads
- Reconcile vendor payments
- Reconcile customer payments

Information

- Documentation
- Blog
- Resource center
- Help center
- Changelog
- Merge for EU

Company

- About us
- Careers

MERGE

Terms of Service | Privacy Policy | GDPR | Cookie settings