



Blog



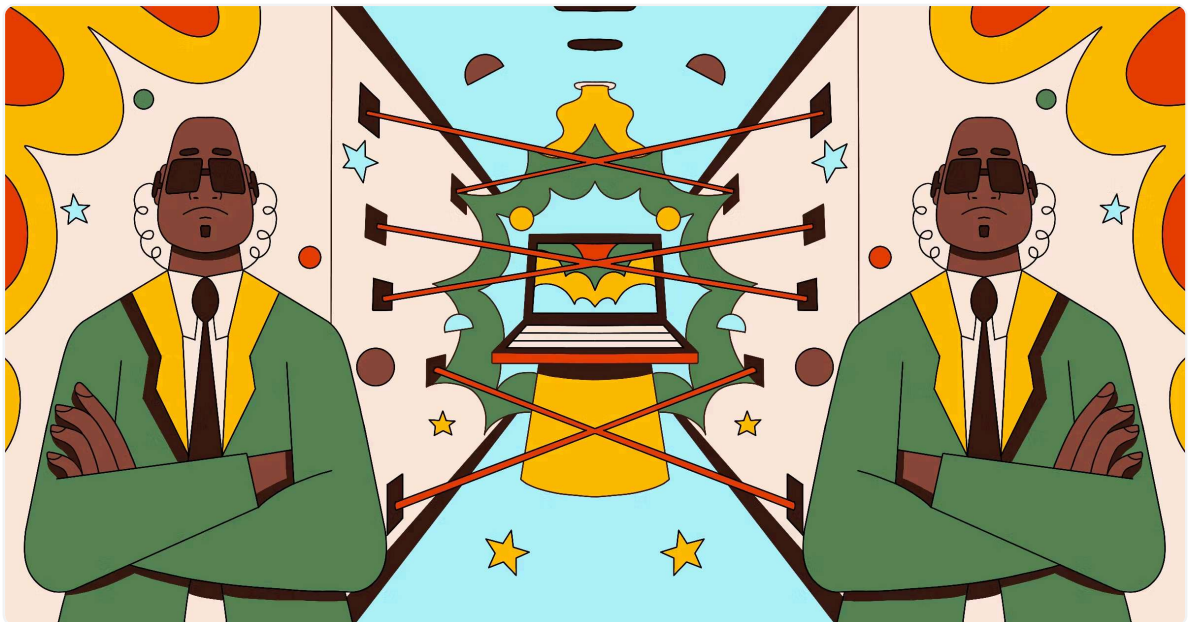
Log in

Sign up

 OCTOBER 6, 2021

Best practices for REST API security: Authentication and authorization

If you have a REST API accessible on the internet, you're going to need to secure it. Here's the best practices on how to do that.



Most apps that use a modern web framework will have one or more [REST APIs](#). [REST is a simple and flexible way](#) of structuring a web API. It's not a standard or protocol, but rather a set of architectural constraints.

There are three reasons you might find yourself writing a REST API:

- To give a networked client that you built—for instance, a [single-page app](#) in the browser or on a mobile app on a phone—access to data

on your server.

- To give end users, both people and programs, programmatic access to data managed by your application.
- To let the many services that make up your app's infrastructure communicate with each other.

Any API built for these reasons can be abused by malicious or reckless actors. Your app will need an access policy—who can view or modify data on your server? For instance, only the author [*Editor's note: the editors, too*] of a blog post should be able to edit it, and readers should only be able to view it. If anyone could edit the post you're reading, then we'd get vandals, link farmers, and others changing and deleting things willy nilly.

This process of defining access policies for your app is called authorization. In this article, we'll show you our best practices for implementing authorization in REST APIs.

Always use TLS

Every web API should use TLS (Transport Layer Security). TLS protects the information your API sends (and the information that users send to your API) by encrypting your messages while they're in transit. You might know TLS by its predecessor's name, SSL. You'll know a website has TLS enabled when its URL starts with `https://` instead of `http://`.

Without TLS, a third party could intercept and read sensitive information in transit, like API credentials and private data! That undermines any of the authentication measures you put in place.

TLS requires a certificate issued by a certificate authority, which also lets users know that your API is legitimate and protected. Most cloud providers and hosting services will manage your certificates and enable TLS for you. If you host a website on Heroku, enabling TLS is a matter of clicking a button. If you host on AWS, AWS Certificate Manager combined with AWS Cloudfront will take care of you. If you can, let your host manage your

certificates for you—it means no hassle at all and every API call will be automatically secured.

If you're running your own web server without any third-party services, you'll have to manage your own certificates. The easiest way to do this is with Let's Encrypt, an automated certificate authority. Let's Encrypt has a helpful [getting started guide](#).

Use OAuth2 for single sign on (SSO) with OpenID Connect

Nearly every app will need to associate some private data with a single person. That means user accounts, and that means logging in and logging out. In the past, you may have written login code yourself, but there's a simpler way: [use OAuth2](#) to integrate with existing single sign-on providers (which we'll refer to as "SSO").

SSO lets your users verify themselves with a trusted third party (like Google, Microsoft Azure, or AWS) by way of token exchange to get access to a resource. They'll log in to their Google account, for instance, and be granted access to your app.

Using SSO means that:

- You don't have to manage passwords yourself! This reduces the user data you store and therefore less data to be exposed in the event of a data breach.
- Not only do you avoid implementing login and logout, but you also avoid implementing multi-factor authentication.
- Your users don't need a new account and new password—they've already got an account with an SSO provider like Google. Less friction at signup means more users for you.

OAuth2 is a standard that describes how a third-party application can access data from an application on behalf of a user. OAuth2 doesn't directly handle authentication and is a more general framework built

primarily for *authorization*. For example, a user might grant an application access to view their calendar in order to schedule a meeting for you. This would involve an OAuth2 interaction between the user, their calendar provider, and the scheduling application.

In the above example, OAuth2 is providing the mechanism to coordinate between the three parties. The scheduling application wants to get an access token so that it can fetch the calendar data from the provider. It obtains this by sending the user to the calendar provider at a specific URL with the request parameters encoded. The calendar provider asks the user to consent to this access, then redirects the user back to the scheduling application with an authorization code. This code can be exchanged for an access token [Here's a good article on the details of OAuth token exchange..](#)

You can implement authentication on top of OAuth2 by fetching information that uniquely identifies the user, like an email address.

However, you should prefer to use OpenID Connect. The OpenID Connect specification is built on top of OAuth2 and provides a protocol for authenticating your users. [Here's a getting started guide on OAuth2 with OpenID Connect.](#)

Unfortunately, not every identity provider supports OpenID Connect. GitHub, for instance, won't let you use OpenID Connect. In that case, you'll have to deal with OAuth2 yourself. But good news—there's an OAuth2 library for your programming language of choice and plenty of good documentation!

Tips for OAuth

You can use OAuth2 in either *stateless* or *stateful* modes. [Here's a good summary on the differences.](#)

The short version: it is typically easier to *correctly* implement a stateful backend to handle OAuth flows, since you can handle more of the sensitive data on the server and avoid the risk of leaking credentials. However, REST APIs are meant to be stateless. So if you want to keep the backend this

way, you either need to use a stateless approach or add an additional stateful server to handle authentication.

If you opt to implement the stateless approach, make sure to use its [Proof Key for Code Exchange](#) mode, which prevents cross-site request forgery and code injection attacks.

You'll need to store users' OAuth credentials. Don't put them in local storage—that can be accessed by any JavaScript running on the page! Instead, store tokens as secure cookies. That will protect against cross-site scripting (XSS) attacks. However, cookies can be vulnerable to cross-site request forgery (CSRF), so you should make sure your cookies use [SameSite=Strict](#).

Use API keys to give existing users programmatic access

While your REST endpoints can serve your own website, a big advantage of REST is that it provides a standard way for other programs to interact with your service. To keep things simple, don't make your users do OAuth2 locally or make them provide a username/password combo—that would defeat the point of having used OAuth2 for authentication in the first place. Instead, keep things simple for yourself and your users, and issue API keys. Here's how:

1. When a user signs up for access to your API, generate an API key: `var token = crypto.randomBytes(32).toString('hex');`
2. Store this in your database, associated with your user.
3. Carefully share this with your user, making sure to keep it as hidden as possible. You might want to show it only once before regenerating it, for instance.
4. Have your users provide their API keys as a header, like `curl -H "Authorization: apikey MY_APP_API_KEY" https://myapp.example.com`

5. To authenticate a user's API request, look up their API key in the database.

When a user generates an API key, let them give that key a label or name for their own records. Make it possible to later delete or regenerate those keys, so your user can recover from compromised credentials.

Encourage using good secrets management for API keys

It's the user's responsibility to keep their secrets safe, but you can also help! Encourage your users to follow best practices by writing good sample code. When showing API examples, show your examples using environment variables, like `ENV["MY_APP_API_KEY"]`.

```
from requests.auth import HTTPBasicAuth
import requests
import os

api_key = os.environ.get("MY_APP_API_KEY")
auth = HTTPBasicAuth('apikey', api_key)
req = requests.get("<https://myapp.example.com>",
                    headers={'Accept': 'application/json'},
                    auth=auth)
```

(If you, like Stripe, write interactive tutorials that include someone's API key, make sure it's a key to a test environment and never their key to production.)

Choose when to enforce authorization with request-level authorization

We've been speaking about API authorization as if it will apply to every request, but it doesn't necessarily need to. You might want to add *request-level authorization*: looking at an incoming request to decide if the user has access to your resources or not. That way, you can let everyone see resources in `/public/`, or choose certain kinds of requests that a user needs to be authenticated to make.

The best way to do this is with request middleware. Kelvin Nguyen over at Caffeine Coding has a nice example [here](#).

Configure different permissions for different API keys

You'll give users programmatic API access for many different reasons. Some API endpoints might be for script access, some intended for dashboards, and so on. Not every endpoint will need the user's full account access. Consider having several API keys with different permission levels.

To do this, store permissions in the database alongside the API keys as a list of strings. Keep this simple at first: "read" and "write" are a great start! Then, add a request middleware that fetches the user and the permissions for the key they've submitted and checks the token permissions against the API.

Leave the *rest* of the authorization to the app/business logic

Now that you've started adding authorization to your API, it can be tempting to add more and more logic to handle more checks. You may end up with nested if-statements for each resource and permission level. The problem with that is that you may end up duplicating application logic. You'll find yourself fetching database records in the middleware, which is not ideal!

Instead, leave that level of authorization logic to your application code. Any authorization checks made on resources should happen in the app, not in the middleware. If you need to handle complex authorization logic in your app, use a tool like [Oso](#), which will let you reduce your authorization policy to a few simple rules.

There's always more to discuss with authentication and authorization, but that's enough to get started! We hope these tips help you design useful and secure API endpoints.

In summary: use good libraries

We've given you plenty of specific advice, but it all comes back to one point—try to offload as much work as you can to trusted libraries. Authorization is tricky, and we'd like to minimize the number of places in which we can make a mistake. You have plenty of great tools at hand to help with authorization, so make the best use of them that you can! Much like with cryptography: study up, and then do as little as possible yourself.

AUTHORS

Sam Scott ›



Graham Neray ›

[API](#)[authentication](#)[authorization](#)[Code for a Living](#)[rest api](#)

RECENT ARTICLES

 JUNE 30, 2025

Reliability for unreliable LLMs

 JUNE 25, 2025

Not an option, but a necessity: How organizations are adopting and implementing AI internally

 JUNE 18, 2025

Learn like a lurker: Gen Z's digital-first lifestyle and the future of knowledge

 JUNE 18, 2025

Smarter teams, brighter insights: Stack Overflow for Teams Business summer bundle

LATEST PODCAST

 JULY 1, 2025

Programming problems that seem easy, but aren't, featuring Jon Skeet

Add to the discussion



Login with your **stackoverflow.com** account to take part in the discussion.

Show **20** comments



Light Dark Auto

Stack Overflow for Teams

Pricing

Use cases

[Customers](#)[Our solution](#)[Integrations](#)[Features](#)[Customer Success](#)[Security](#)[Return on Investment \(ROI\)](#)

OverflowAI

Available on Enterprise.

[Try free](#)[Log in](#)[Engineers](#)[Data Scientists](#)[DevOps & SRE](#)[Support](#)[Product Management](#)

Resources

[Productivity](#)[AI/ML](#)[Guides and Insights](#)[Customer Academy](#)[FAQ](#)[Help center](#)

Stack Overflow Advertising

[Why Stack Overflow?](#)[What to expect](#)[Advertise to developers](#)[Attract tech talent](#)[Post a job !\[\]\(eff7520f80aa06fb7298beb68337d76d_img.jpg\)](#)

Powered by Indeed

Use cases

[Marketing Teams](#)[Employer Branding Teams](#)[DevRel Teams](#)[Talent Teams](#)[Technology Teams](#)[Agencies](#)

Resources

[Product guides & insights](#)[Community insights](#)[Advertising best practices](#)[Talent best practices](#)

Company

Knowledge Solutions

Data licensing offering to build and improve AI tools and models.

[About](#)[Leadership](#)[Social Impact](#)[Press](#)

Careers

Open positions

Contact us

Partnerships

Blog

Newsletter

Podcast

Labs


Annual Developer Survey

Site design / logo © 2025 Stack Exchange Inc.

[Terms](#)

[Privacy policy](#)

[Cookie policy](#)

[Your Privacy Choices](#) 

[Go to stackoverflow.com](#)

