TUTORIAL

Build a REST API in Node.js with Express and JWT Auth

Learn how to build a practical REST API in Node.js with Express and JWT authentication, including setup, protected routes, validation, testing, and production readiness checks.

Programming - July 19, 2026

What you will build

A REST API becomes operationally useful only when it has a clear request model, predictable error handling, and a way to protect sensitive endpoints. In this tutorial, you will build a small Node.js API with Express and JWT authentication that includes a health check, a login endpoint, and one protected route. By the end, you will know how to issue a token, send it back on requests, verify it in middleware, and confirm that the API behaves correctly before production use.

The finished state is intentionally modest: a maintainable API skeleton that you can extend with real user storage, authorization rules, rate limiting, and hardened token handling. If you only need public endpoints, JWT may be unnecessary; if you need authenticated access without session state on the server, this pattern is a practical fit. For deeper token-design considerations, see How to Secure Node.js APIs with JWT Authentication and Secure Node.js API Authentication with JWT and OAuth 2.0.

Prerequisites and stop-here checks

Goal

Confirm that the runtime, package manager, and security assumptions are valid before you write code. JWT-based APIs are straightforward to build, but they are easy to misuse if secrets are weak, token expiry is ignored, or you plan to store sensitive data in the token itself.

Action

You should have:

- Node.js 18 or later installed
- npm available
- A terminal and a code editor
- A local development environment where you can run environment variables safely
- A clear decision on how user credentials will be verified, even if the first version uses a simple in-memory check

Stop here if you do not yet know where the signing secret will come from in each environment. Do not hardcode the JWT secret in source control. Also stop here if you plan to use this pattern for high-risk authorization without defining claim validation, expiry, and key rotation expectations.

Expected output

A development setup where you can create a project, install dependencies, and run the API locally without exposing secrets.

Validation

Run:

Confirm that both commands succeed and that your Node.js version is current enough for your project policies.

Common failure

- Using an outdated runtime that breaks package installation or modern syntax
- Storing secrets directly in code or committing them to version control
- Treating a demo login flow as production authentication without later replacement

Prepare the project

Goal

Create a minimal Express application structure with the dependencies needed for routing, JSON parsing, and token signing.

Action

Create a new project directory and initialize it:

Then create a simple structure:

Add scripts to package.json:

Create a .env file for development only:

Expected output

A runnable Node.js project with Express, JWT support, environment-based configuration, and a development reload workflow.

Validation

Check that dependencies installed successfully and that npm run dev can start the app once server.js exists.

Common failure

- Forgetting to add dotenv, which causes process.env values to be undefined
- Using a short or guessable secret
- Running a development watcher in production by accident

Build the Express server

Goal

Create the API server, enable JSON request parsing, and add a basic endpoint that proves the service is alive.

Action

Create src/server.js:

Expected output

The server starts and exposes GET /health, returning a small JSON response.

Validation

Start the API and test it:

Expected response:

Common failure

- Missing express.json(), which breaks JSON body parsing on POST requests
- Incorrect route mounting, which causes /auth paths not to resolve
- Port conflicts if another process is already listening on the chosen port

Implement token issuance

Goal

Create a login endpoint that validates credentials and returns a signed JWT when authentication succeeds.

Action

Create src/auth.js:

This example uses a fixed demo user to keep the implementation focused. In a real system, credential checks should query your identity source, compare hashed passwords, and enforce lockout or throttling rules where appropriate.

Expected output

A /auth/login endpoint that returns a JWT when the supplied credentials match the expected user.

Validation

Send a login request:

Expected response includes an accessToken field with a long encoded string.

Common failure

- Missing request body JSON
- Secret not loaded from .env
- Using a static demo password beyond local development

Protect routes with JWT middleware

Goal

Verify the token on protected requests and reject unauthenticated calls early and consistently.

Action

Create src/middleware/authenticateToken.js:

Update src/server.js with one protected endpoint:

Expected output

Requests to /profile fail without a valid token and succeed when the client sends Authorization: Bearer <token>.

Validation

Without a token:

Expected response:

With a token copied from /auth/login:

Expected response includes message and the decoded user claim.

Common failure

- Accepting tokens from the wrong header format
- Verifying tokens with a different secret than the one used to sign them
- Returning detailed verification errors that expose too much implementation detail

Decide what belongs in the token

Goal

Keep JWT payloads small and safe to expose to clients, because the token is only signed, not encrypted.

Action

Limit the payload to claims needed for request processing, such as subject, username, and token expiry. Avoid putting passwords, secrets, or unnecessary personal data into the token. If downstream services need roles or tenant identifiers, add only the minimal claims required and document how they are validated.

Expected output

A token that is sufficient for authorization decisions without carrying sensitive or bloated data.

Validation

Inspect the encoded payload after decoding the token and confirm that it contains only intended claims.

Common failure

- Treating JWTs as a storage mechanism for sensitive data
- Allowing claim sprawl without a validation policy
- Depending on unverified client-side claims for server-side authorization

Validate the API end to end

Goal

Confirm the request flow works from login through protected access and that failure cases behave as expected.

Action

Test these cases in order:

- Login with valid credentials and confirm token issuance.
- Call /profile with no token and confirm a 401 response.
- Call /profile with an expired or invalid token and confirm a 401 response.
- Call /profile with a valid token and confirm a 200 response.

For a quick manual check, you can use curl or a REST client. If you prefer an automated check, write a small integration test that starts the app, gets a token, and asserts the protected route response.

Expected output

A reproducible validation path showing that authentication is enforced and that failures are safe.

Validation

A healthy implementation should demonstrate:

- 200 OK for successful login
- 401 Unauthorized for missing or invalid bearer tokens
- A small, predictable response body for the protected endpoint

Common failure

- A protected route accidentally left public
- Token expiry not being enforced because verification is skipped
- Accepting malformed authorization headers

Operational follow-up before production use

Goal

Turn the tutorial implementation into something safer to operate in a real environment.

Action

Review these controls before production:

- Replace the demo user with real identity lookup and password hashing
- Store JWT_SECRET in a secrets manager or environment injection mechanism, not in source
- Set an expiry appropriate to your risk model and session requirements
- Decide whether you need refresh tokens, revocation, or key rotation
- Add request logging and rate limiting to reduce brute-force and abuse risk; if that is part of your deployment plan, Node.js Rate Limiting with Redis: Secure API Throttling explains a practical pattern
- Ensure every protected route uses the middleware consistently
- Verify that error messages do not disclose token internals or credential hints

Expected output

A clear production checklist for the API, with the remaining gaps understood before deployment.

Validation

Before release, confirm:

- Secrets are injected securely
- Token expiry is set and tested
- Unauthorized requests return 401
- Logs do not expose tokens
- Authentication and authorization responsibilities are separated in code

Common failure

- Deploying with demo credentials still enabled
- Reusing one secret across environments without rotation plans
- Assuming authentication alone is enough when authorization rules are still undefined

Final takeaway

You now have a working Express API with a JWT login flow, middleware-based route protection, and a validation path you can repeat locally. The implementation is intentionally small, but the operational discipline matters: verify the token format, protect every sensitive route, keep claims minimal, and confirm that secrets and expiry settings are correct before production use. If you extend this pattern carefully, the same structure can support a more complete API without turning authentication into an afterthought.

Use this guidance together with git branching checklist and branch and bound algorithm to connect the workflow with related operational context already available on the site.