Nextjs API Best Practice 2025

6 min read · Feb 3, 2025

Lior Amsalem Follow

Listen Share

When building APIs in **Next.js 15**, security and performance are critical. Without best practices, APIs are vulnerable to attacks, data loss, and inefficiencies. Below are some **best practices** to follow when designing APIs in Next.js 15. in addition to those best practices if you want to learn how to code faster you should use gpteach, it will greatly improve your coding writing skills and productivity.

1. Authentication & Authorization

What is Authentication?

Authentication verifies **who is making the request**. This ensures that only valid users can access your API. Common authentication methods include:

- **JWT (JSON Web Tokens):** A token-based authentication method where users log in once and receive a signed token to include in future API requests.
- **OAuth:** A delegated authentication protocol (e.g., logging in via Google).
- **Session-Based Authentication:** A traditional method where user sessions are stored on the server.

Example: JWT Authentication in Next.js API (`route.ts` **)**

Code below is just an general exmplae, you'll use something different in your app or when you use 3rd party you might used thier feature.

```
import { NextRequest, NextResponse } from "next/server";
import jwt from "jsonwebtoken";

const SECRET_KEY = process.env.JWT_SECRET!;

export async function GET(request: NextRequest) {

  const token = request.headers.get("Authorization")?.split(" ")[1]; // Extract
  if (!token) return NextResponse.json({ message: "Unauthorized" }, { status: 4

  try {
    const decoded = jwt.verify(token, SECRET_KEY); // Verify JWT
    return NextResponse.json({ message: "Authenticated", user: decoded });
  } catch {
    return NextResponse.json({ message: "Invalid token" }, { status: 401 });
  }
}
```

What is Authorization?

Authorization determines **what resources a user can access** after authentication.

For example:

- **Admins** can delete and create posts.
- **Members** can read and comment on posts.
- **Guests** can only view public content.

Example: Role-Based Authorization

Role allow us to restrict the request owner ability to access certain parts of our data, an admin would have access to sensitive data that a member or a manager won't have. sometime authorization is simplify a business logic in our application, where in certain cases user can't place data and it need to be restrict or authorized to do so.

```
export async function GET(request: NextRequest) {
  const token = request.headers.get("Authorization")?.split(" ")[1];

 if (!token) return NextResponse.json({ message: "Unauthorized" }, { status: 40
  try {
    const decoded: any = jwt.verify(token, SECRET_KEY);
    if (decoded.role !== "admin") {
      return NextResponse.json({ message: "Forbidden" }, { status: 403 });
    }
    return NextResponse.json({ message: "Admin Access Granted" });
  } catch {
    return NextResponse.json({ message: "Invalid token" }, { status: 401 });
  }
}
```

2. Rate Limiting

Why Use Rate Limiting?

Rate limiting prevents **abuse and excessive API calls.** Without it, attackers or scrapers could:

- **Download all public content** (e.g., steal all your blog posts).
- **Overload the server** (Denial of Service attacks).
- **Exploit brute-force login attempts.**

How to Implement Rate Limiting in Next.js?

Using **next-rate-limit:**

```
import { NextRequest, NextResponse } from "next/server";
import { RateLimiterMemory } from "rate-limiter-flexible";

const rateLimiter = new RateLimiterMemory({ points: 5, duration: 60 }); // 5 re
export async function GET(request: NextRequest) {
  try {
    await rateLimiter.consume(request.ip || "anonymous"); // Consume request
    return NextResponse.json({ message: "Request allowed" });
  } catch {
    return NextResponse.json({ message: "Too many requests" }, { status: 429 })
    }
  }
```

This **limits each user** to **5 requests per minute**, reducing server overload and API scraping.

3. Input Validation & Sanitization

Data arrives from sources we can't verify, even if it's written by us, even if it's from our own application, we can't verify the source of data. This is why we always want to validate data, we always want to make sure data is as it was expected to be receive and reject it if it's not. we also want to sanitise the data, remove unexpected things or delete or trim or remove things we consider malicious (like HTML in place that shouln't have HTML etc)

Why Sanitize Data?

Without sanitization, APIs are vulnerable to:
SQL Injection — Malicious queries injected into database queries.
XSS (Cross-Site Scripting) — Injecting malicious scripts into web pages.
HTML Injection — Users inserting raw HTML into inputs.

How to Secure Incoming Data?

Example: Validate & Sanitize User Input

Using **zod** to validate input will make sure the data we receive from external source (even if we wrote the request ourself and it's from "our own application") the validation process will make sure the data in our production will not be compromised be bad data (e.g forcing string into a number column in database or very big content into a small column etc)

```
import { z } from "zod";
import { NextRequest, NextResponse } from "next/server";

const userSchema = z.object({
  name: z.string().min(3).max(50),
  email: z.string().email(),
});
```

```
export async function POST(request: NextRequest) {
  const body = await request.json();
  const parsed = userSchema.safeParse(body);
  if (!parsed.success) {
    return NextResponse.json({ message: "Invalid data" }, { status: 400 });
  }
  return NextResponse.json({ message: "Valid input", user: parsed.data });
}
```

Prevents invalid data from reaching the database.
Protects against **code injection** attacks.

NOTE: might those are not the best tools for your job, maybe you'll and should choose something else other than zod! however you should validate data before you push it into your database!

4. Avoid Exposing Sensitive Information

Do not return sensitive data in API responses:

Bad Practice

```
return NextResponse.json({ message: "Success", user });
```

The reason above is a bad practice is because down the road we mgiht store additional data on the user object, which means that data, even if it's sensitive (for example password, or salt, or prviate information) it will be automaticly added to our API without us even expexting it to be expose like that! this is why we want to be explicit with our API.

Good Practice

The good practice allow us to be explicit about what we want to expose to the users, hence making user that we don't send to the public domain of the internet sensitive data.

```
return NextResponse.json({ message: "Success", user: { id: user.id, name: user.
```

5. Secure Database Queries

Use **parameterized queries** to prevent **SQL Injection:**

```
import { sql } from "@vercel/postgres";

export async function GET(request: NextRequest) {
  const { searchParams } = new URL(request.url);
  const id = searchParams.get("id");
  if (!id) return NextResponse.json({ message: "Missing ID" }, { status: 400 })
  const user = await sql`SELECT * FROM users WHERE id = ${id}`; // Secure query
  return NextResponse.json(user);
}
```

6. Use Environment Variables for Secrets

In many application we'll find ourself with data and information that is meant to be saved as a secret, it' might be meant to be used on the server side and never be exposed to 3rd parties or it might meant to be used when we hash data before we expose it. in some cases the data is not very sensitive but we still want to be on the safe side, this is why we'll want to store sensitive data in `.env` files instead of hardcoding them:

Access variables in code:

```
const secretKey = process.env.JWT_SECRET;
```

7. Use CORS to Restrict API Access

CORS (Cross-Origin Resource Sharing) controls who can access your API.

With CORS we want to be strict about where we get the request from, if we don't expect to integrate with external resoucres or external 3rd party API than we should heavily consider limiting who can submit data to us. Hence why we'll choose to allow only specific origins.

```
import { NextRequest, NextResponse } from "next/server";

export function middleware(req: NextRequest) {
  const allowedOrigin = "https://yourdomain.com";
  if (req.headers.get("origin") !== allowedOrigin) {
    return NextResponse.json({ message: "CORS blocked" }, { status: 403 });
  }
  return NextResponse.next();
}
```

Conclusion

By following these **best practices**, you can build a **secure, scalable, and maintainable** Next.js 15 API:

Next.js API Best Practices Recap

Authentication & Authorization — Verify users and control access.
Rate Limiting — Prevent abuse and excessive API calls.
Input Sanitization — Clean data to prevent injections and security risks.
Avoid Exposing Sensitive Data — Do not return passwords or private info.
Secure Database Queries — Use parameterized queries.
Environment Variables — Store secrets securely.
Use CORS — Restrict API access to trusted domains.

Following these practices ensures **better security, performance, and maintainability** for your Next.js 15 APIs. Happy coding!

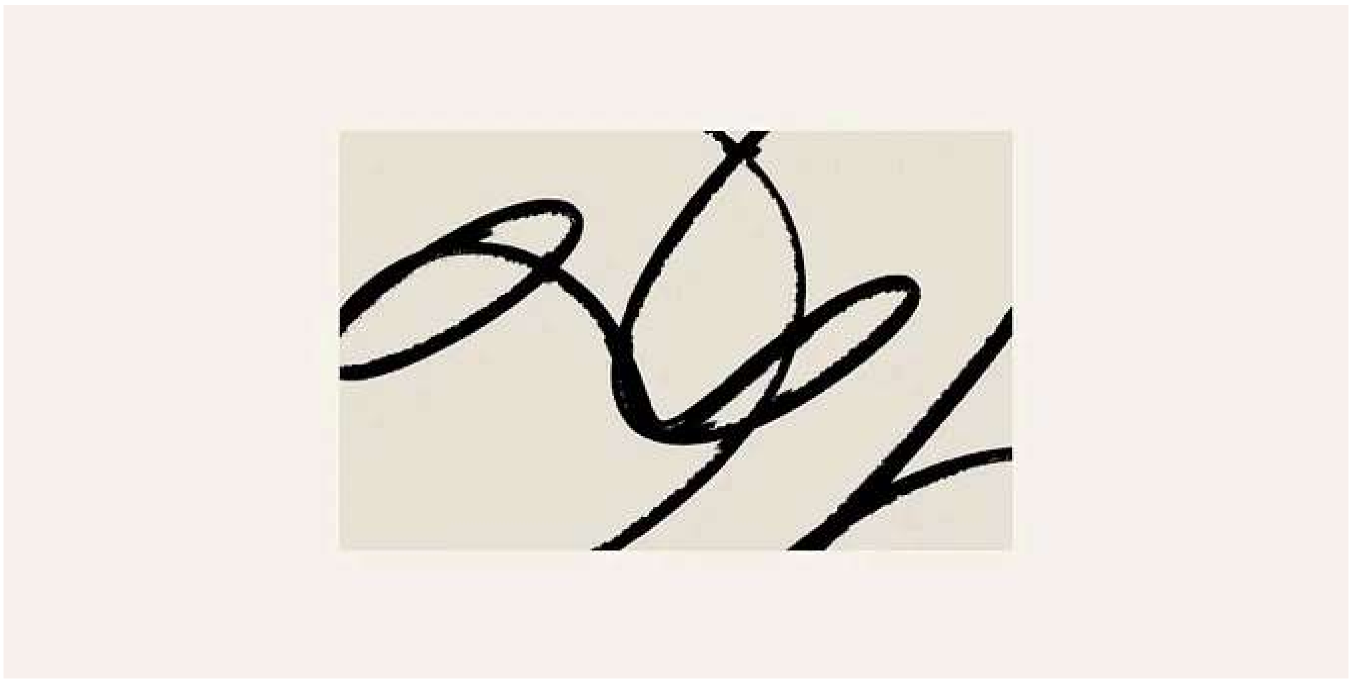
Best Practice JavaScript Coding Skills Writing Code

Follow

Written by Lior Amsalem

79 followers · 64 following

More from Lior Amsalem

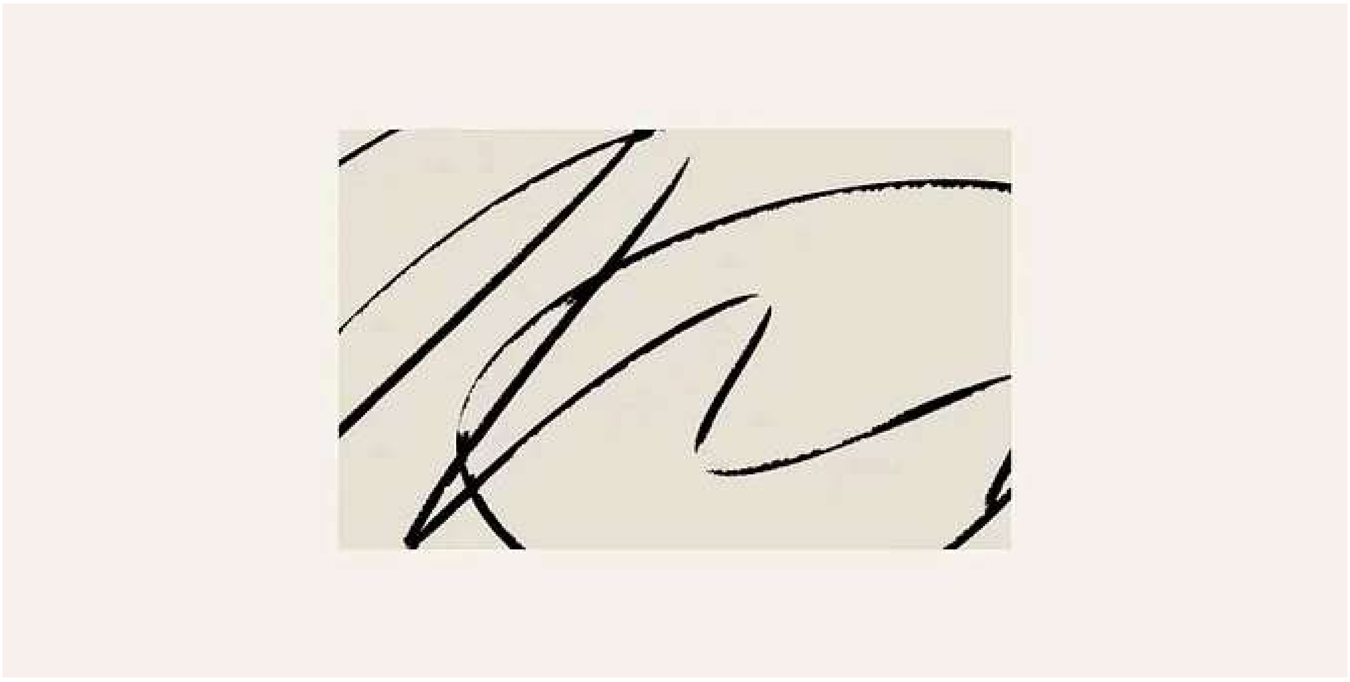


Lior Amsalem

3 Biggest Mistakes with Drizzle ORM

In this article I want to showcase 3 biggest mistakes with drizzle ORM and also in addition I want to share my biggest mistake with my...

Feb 9 75 1

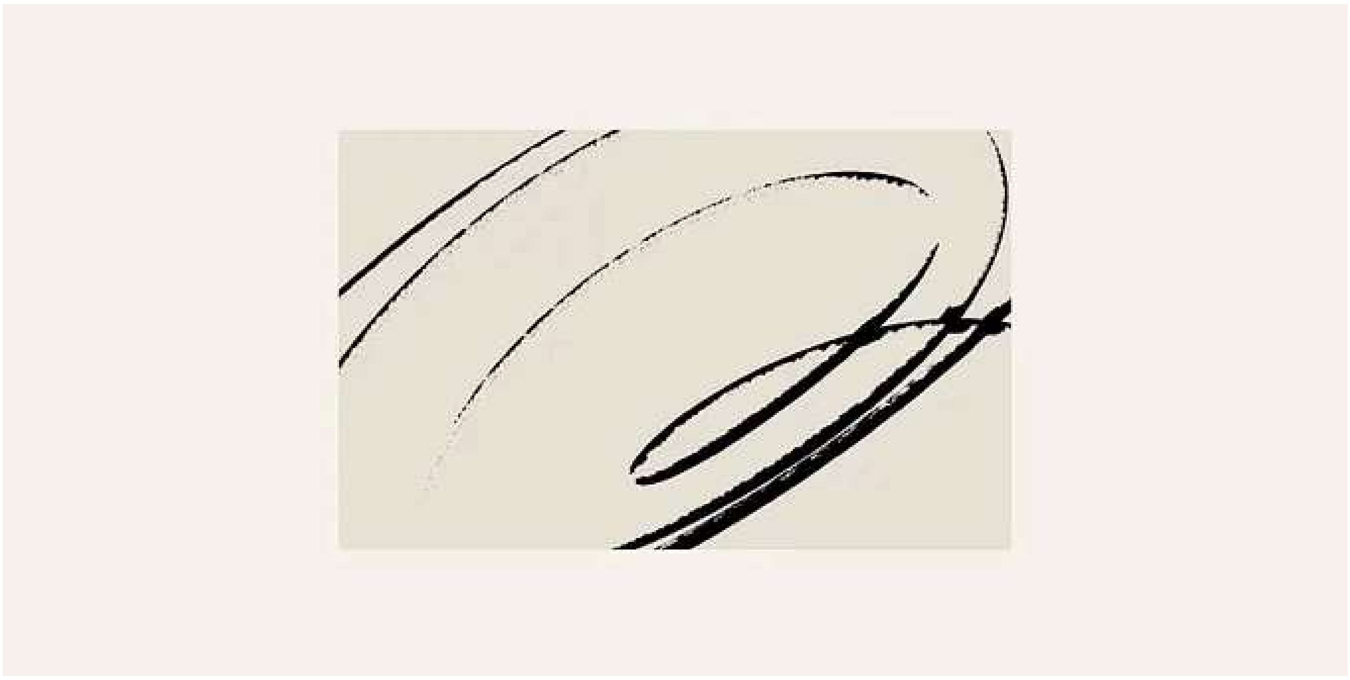


Lior Amsalem

Building an API with FastAPI and Supabase

Before we jump to basic implementation we'll go over what the basics are of what supabase is, what fastapi is and also I want to encourage...

Feb 16 7 2

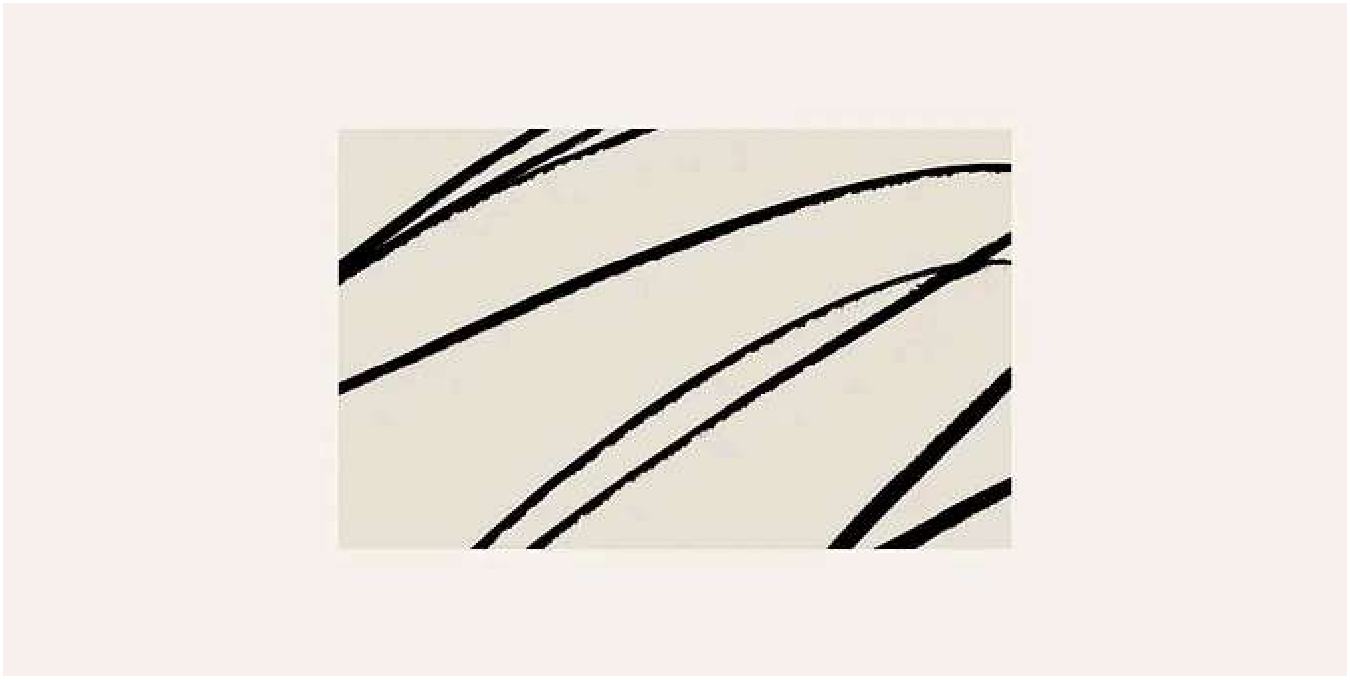


Lior Amsalem

ENUM with TypeScript, Zod AND Drizzle ORM

Enums are a powerful way to define a set of possible values for a field. When working with Drizzle ORM, you can use the pgEnum to define...

Jan 15 7 2



Lior Amsalem

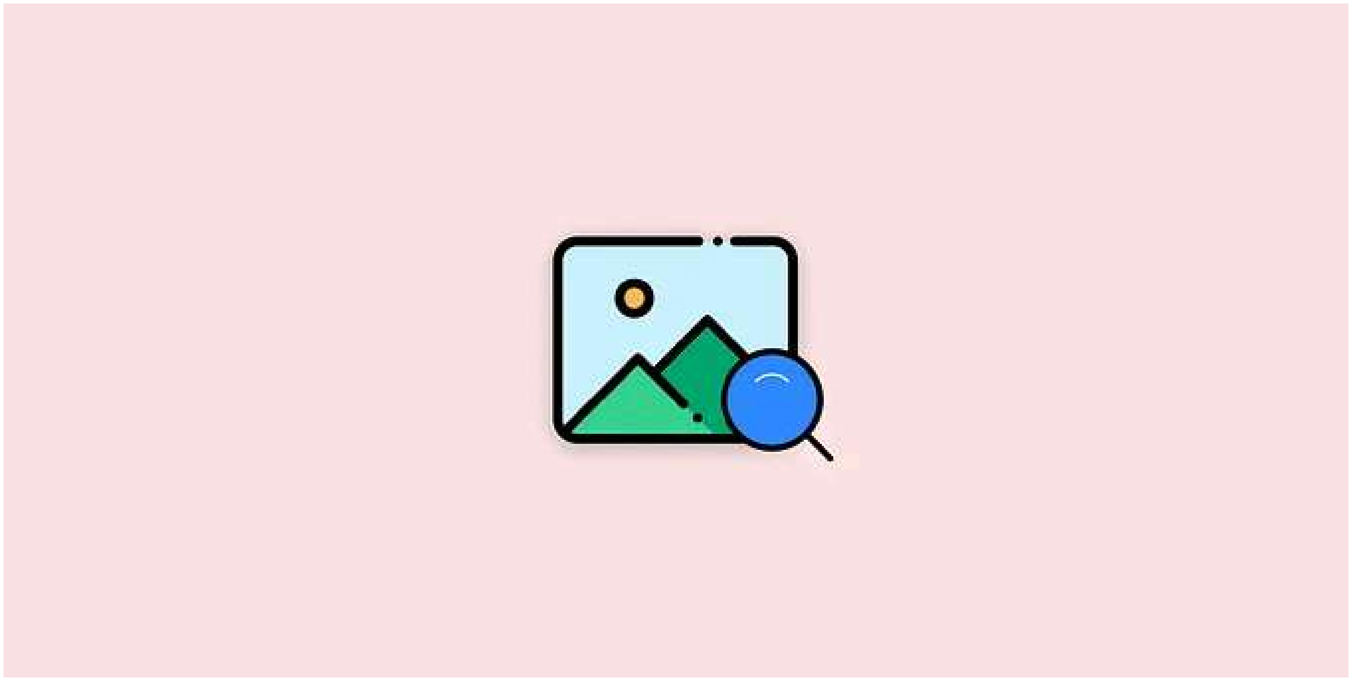
3 Biggest Mistakes Using Supabase

Supabase is a scale up postgresql database provider, which means that if you want to grow fast, it's good for you. in addition supabase...

Feb 8 4 1

See all from Lior Amsalem

Recommended from Medium



In JavaScript in Plain English by Meet

5 Common Mistakes Developers Make with Next.js Image Optimization (And How to Fix Them)

Read for free: Click Here

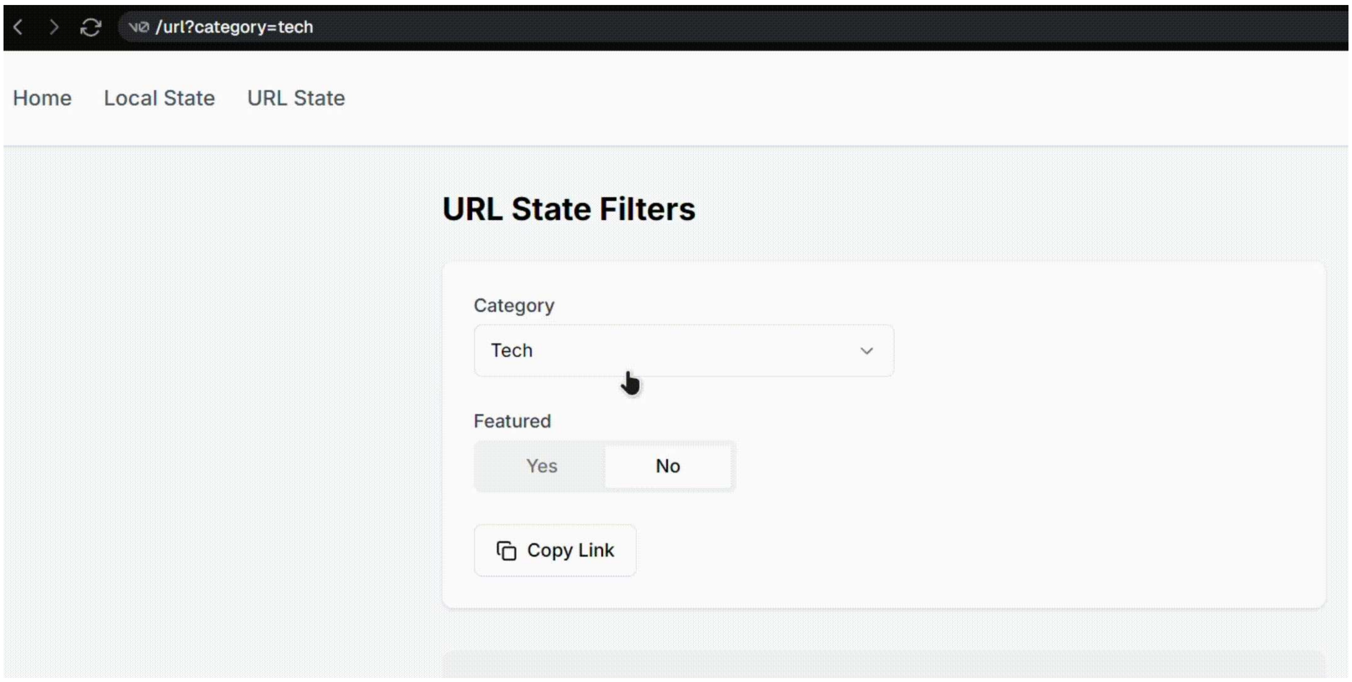
Sep 2 1

Abrar Hussain : Software Engineer

Next.js Function/Next.js getStaticPaths() Function

Next.js getStaticPaths() Function

Mar 19 1 1



In devdotcom by Subinoy Biswas

useState Is Not Enough: The URL Is Your New State Manager (NextJS)

Still using useState for filters? That's cute… until your users hit refresh and lose everything.

Apr 21 17

Aman Shekhar

Next.js has gained immense popularity as a framework for building React applications due to its…

Understanding the Next.js Architecture

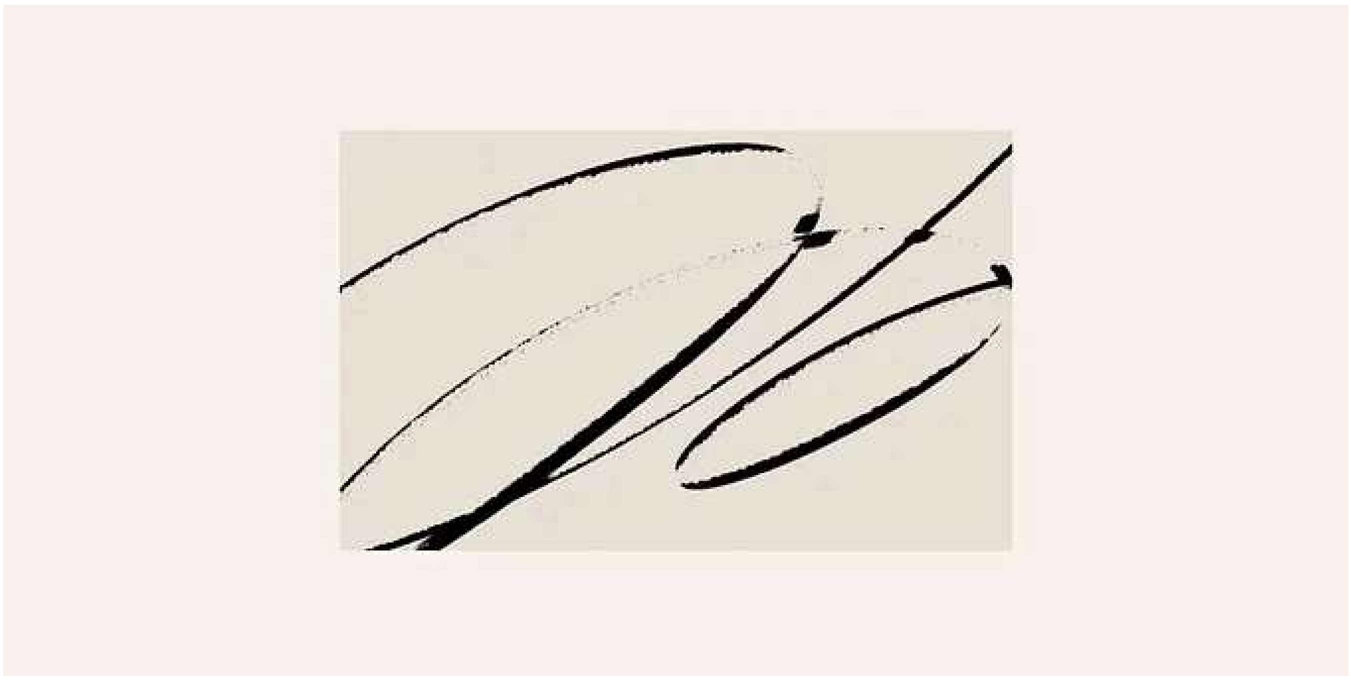
Sep 2

habtesoft

The Secret HTML Tags Top Developers Use

When most developers think of HTML, they think of the usual suspects: <div>, <span>, <p>, <h1>, and <a>. But HTML has a treasure trove of…

Sep 7 76



Turingvang

Nextjs API Calls 2025

Let’s start with basic of nextjs, Next.js is a modern React framework for building full-stack web applications. It extends React with…

May 1 4 1

See more recommendations