Setting Up a Live API Shield Demo with Cloudflare Workers (Windows)

Overview & Strategy

You'll build a tiny "PeakCart" product catalog API on Workers, route it through a zone you control, upload an OpenAPI schema to API Shield, then demonstrate schema validation live by sending both valid and invalid requests. The whole setup takes about 30–45 minutes and costs nothing (Workers free tier + your Enterprise zone for API Shield).

All commands below are for **PowerShell** (the default terminal in Windows Terminal and VS Code on Windows).

Prerequisites

- A Cloudflare account with Enterprise access
- A domain added to Cloudflare (even a cheap `.dev` or `.xyz` works) **or** you can use the default `*.workers.dev` subdomain
- Node.js 18+ installed locally — download from nodejs.org or install via `winget install OpenJS.NodeJS.LTS`
- Wrangler CLI: `npm install -g wrangler`
- Logged in: `wrangler login`

Step 1: Create the Worker (Your Fake "PeakCart" API)

1a. Scaffold the project

```powershell
mkdir peakcart-api
cd peakcart-api
npm init -y
npm install wrangler --save-dev
```

Create `wrangler.toml` (you can use VS Code, Notepad, or PowerShell):

```powershell
```

```
# Quick way to create the file from PowerShell:
@"
name = "peakcart-api"
main = "src/index.js"
compatibility_date = "2024-12-01"

# If you have a custom domain, add a route:
# routes = [
#   { pattern = "api.yourdomain.com/*", zone_name = "yourdomain.com" }
# ]
"@ | Set-Content -Path wrangler.toml -Encoding UTF8
```

1b. Write the API

```powershell
mkdir src
```

Create `src/index.js` — open it in your editor, or use PowerShell:

```powershell
```

```
@"
// PeakCart Demo API — Product Catalog
// This simulates the kind of API a headless commerce platform exposes.

const PRODUCTS = [
  { id: 1, name: "Wireless Headphones", price: 79.99, category: "electronics", stock: 142 },
  { id: 2, name: "Running Shoes", price: 129.95, category: "footwear", stock: 58 },
  { id: 3, name: "Organic Coffee Beans", price: 18.50, category: "grocery", stock: 300 },
  { id: 4, name: "Leather Backpack", price: 199.00, category: "accessories", stock: 23 },
  { id: 5, name: "Yoga Mat", price: 34.99, category: "fitness", stock: 87 },
];

export default {
  async fetch(request, env, ctx) {
    const url = new URL(request.url);
    const path = url.pathname;
    const method = request.method;

    // CORS headers for testing from browser
    const corsHeaders = {
      "Access-Control-Allow-Origin": "*",
      "Access-Control-Allow-Methods": "GET, POST, OPTIONS",
      "Access-Control-Allow-Headers": "Content-Type",
    };

    if (method === "OPTIONS") {
      return new Response(null, { headers: corsHeaders });
    }

    // GET /api/v1/products — List all products
    if (path === "/api/v1/products" && method === "GET") {
      const category = url.searchParams.get("category");
      let results = PRODUCTS;
      if (category) {
        results = PRODUCTS.filter(p => p.category === category);
      }
      return Response.json(
        { success: true, data: results, count: results.length },
        { headers: corsHeaders }
      );
    }

    // GET /api/v1/products/:id — Get single product
```

```
const singleMatch = path.match(/^\/api\/v1\/products\/(\d+)$/);
if (singleMatch && method === "GET") {
  const id = parseInt(singleMatch[1]);
  const product = PRODUCTS.find(p => p.id === id);
  if (!product) {
    return Response.json(
      { success: false, error: "Product not found" },
      { status: 404, headers: corsHeaders }
    );
  }
  return Response.json(
    { success: true, data: product },
    { headers: corsHeaders }
  );
}

// POST /api/v1/cart/add — Add item to cart
// Expects: { "product_id": int, "quantity": int }
if (path === "/api/v1/cart/add" && method === "POST") {
  let body;
  try {
    body = await request.json();
  } catch (e) {
    return Response.json(
      { success: false, error: "Invalid JSON body" },
      { status: 400, headers: corsHeaders }
    );
  }

  const { product_id, quantity } = body;

  if (!Number.isInteger(product_id) || !Number.isInteger(quantity)) {
    return Response.json(
      { success: false, error: "product_id and quantity must be integers" },
      { status: 400, headers: corsHeaders }
    );
  }

  if (quantity < 1 || quantity > 10) {
    return Response.json(
      { success: false, error: "quantity must be between 1 and 10" },
      { status: 400, headers: corsHeaders }
    );
  }
```

```
      const product = PRODUCTS.find(p => p.id === product_id);
      if (!product) {
        return Response.json(
          { success: false, error: "Product not found" },
          { status: 404, headers: corsHeaders }
        );
      }

      return Response.json(
        {
          success: true,
          message: "Added " + quantity + "x " + product.name + " to cart",
          cart_total: (product.price * quantity).toFixed(2),
        },
        { status: 201, headers: corsHeaders }
      );
    }

    // Health check
    if (path === "/api/v1/health") {
      return Response.json(
        { status: "ok", service: "peakcart-api", version: "1.0.0" },
        { headers: corsHeaders }
      );
    }

    // Catch-all: 404
    return Response.json(
      { success: false, error: "Endpoint not found" },
      { status: 404, headers: corsHeaders }
    );
  },
};
"@ | Set-Content -Path src/index.js -Encoding UTF8
```

1c. Test locally

```powershell
npx wrangler dev
```

Open a **second PowerShell tab** and test. PowerShell has `curl` as an alias for `Invoke-WebRequest`, which behaves differently from Unix curl. Use `Invoke-RestMethod` instead — it automatically parses JSON and gives clean output:

```powershell
# List products
Invoke-RestMethod http://localhost:8787/api/v1/products

# Filter by category
Invoke-RestMethod "http://localhost:8787/api/v1/products?category=electronics"

# Get single product
Invoke-RestMethod http://localhost:8787/api/v1/products/1

# Add to cart (valid)
$validBody = '{"product_id": 1, "quantity": 2}'
Invoke-RestMethod http://localhost:8787/api/v1/cart/add -Method POST -ContentType "application/json" -Body $validBody

# Add to cart (invalid — this is what schema validation will catch)
$invalidBody = '{"product_id": "abc", "quantity": 999, "coupon_code": "HACK"}'
Invoke-RestMethod http://localhost:8787/api/v1/cart/add -Method POST -ContentType "application/json" -Body $invalidBod
```

> **Tip:** If you prefer the Unix-style `curl` syntax, install the real curl via `winget install cURL.cURL`, then invoke it as `curl.exe` (the `.exe` bypasses the PowerShell alias). All the curl commands from the demo section below will then work exactly as written if you use `curl.exe` instead of `curl`.

1d. Deploy

```powershell
npx wrangler deploy
```

Your API is now live at `https://peakcart-api.<your-subdomain>.workers.dev`.

If you want it on a custom domain (looks more professional for the demo), either:

- Add a route in `wrangler.toml` pointing to `api.yourdomain.com/*`
- Or add a Custom Domain in the Workers dashboard: **Workers & Pages > peakcart-api > Settings > Triggers > Custom Domains**

Step 2: Create the OpenAPI Schema

Save this as `peakcart-schema.yaml`. You can create it in VS Code or from PowerShell:

```powershell
```

```
@"
openapi: 3.0.3
info:
  title: PeakCart Product API
  description: E-commerce product catalog and cart API
  version: "1.0.0"
servers:
  - url: https://peakcart-api.YOUR-SUBDOMAIN.workers.dev
    description: Production

paths:
  /api/v1/products:
    get:
      operationId: listProducts
      summary: List all products
      parameters:
        - name: category
          in: query
          required: false
          schema:
            type: string
            enum: [electronics, footwear, grocery, accessories, fitness]
      responses:
        "200":
          description: Product list

  /api/v1/products/{productId}:
    get:
      operationId: getProduct
      summary: Get a single product by ID
      parameters:
        - name: productId
          in: path
          required: true
          schema:
            type: integer
            minimum: 1
      responses:
        "200":
          description: Single product
        "404":
          description: Product not found
  /api/v1/cart/add:
    post:
      operationId: addToCart
      summary: Add a product to the shopping cart
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required:
                - product_id
                - quantity
              properties:
                product_id:
                  type: integer
                  minimum: 1
                  description: The ID of the product to add
                quantity:
                  type: integer
                  minimum: 1
                  maximum: 10
                  description: Number of items to add (max 10)
              additionalProperties: false
      responses:
        "201":
          description: Item added to cart
        "400":
          description: Validation error
        "404":
          description: Product not found

  /api/v1/health:
    get:
      operationId: healthCheck
      summary: API health check
      responses:
        "200":
          description: Service is healthy
"@ | Set-Content -Path peakcart-schema.yaml -Encoding UTF8
```

Important: Replace `YOUR-SUBDOMAIN` in the `servers` URL with your actual workers.dev subdomain (or your custom domain if you set one up).

Step 3: Set Up API Shield in the Dashboard

3a. Upload the schema

1. Log in to the **Cloudflare Dashboard**
2. Select your **domain/zone**
3. Go to **Security > API Shield**
4. Click **Schema Validation** (or in the new UI: **Web assets > Schema validation tab**)
5. Click **Add validation**
6. Upload `peakcart-schema.yaml`
7. Review the detected endpoints — you should see:
   - `GET /api/v1/products`
   - `GET /api/v1/products/{productId}`
   - `POST /api/v1/cart/add`
   - `GET /api/v1/health`
8. Set the action for non-compliant requests to **Log** initially (you'll switch to Block in the demo)
9. Check **"Save new endpoints to endpoint management"**
10. Click **Add schema and endpoints**

3b. Verify endpoints appear in Endpoint Management

1. Go to **Security > API Shield > Endpoint Management** (or **Web assets > Endpoints tab**)
2. You should see all four endpoints listed with their methods
3. Cloudflare will start collecting traffic data immediately

3c. Generate some traffic

Before the demo, send a burst of legitimate traffic so the analytics dashboards have data to show:

```powershell
```

```
# Traffic generation loop — run this and let it go for a few minutes
$baseUrl = "https://peakcart-api.YOUR-SUBDOMAIN.workers.dev"

1..50 | ForEach-Object {
    Invoke-RestMethod "$baseUrl/api/v1/products" | Out-Null
    Invoke-RestMethod "$baseUrl/api/v1/products/1" | Out-Null
    Invoke-RestMethod "$baseUrl/api/v1/products/2" | Out-Null
    Invoke-RestMethod "$baseUrl/api/v1/cart/add" -Method POST -ContentType "application/json" -Body '{"product_id": 1, "
    Invoke-RestMethod "$baseUrl/api/v1/cart/add" -Method POST -ContentType "application/json" -Body '{"product_id": 3, "
    Invoke-RestMethod "$baseUrl/api/v1/health" | Out-Null
    Start-Sleep -Milliseconds 500
    Write-Host "Batch $_ of 50 sent"
}
```

Step 4: The Demo Flow — What to Show Live

> **For the live demo commands below, use `curl.exe`** (the real curl, not the PowerShell alias). Windows 10/11 ships with it at `C:\Windows\System32\curl.exe`. Using `curl.exe` means you get the same familiar output the audience expects to see — HTTP headers, response bodies, status codes — without PowerShell's formatting getting in the way.

Part A: "Here's what your API surface looks like"

Open **API Shield > Endpoint Management**. Point out:

- "These are the endpoints API Shield has identified. In your environment, Cloudflare's ML-based discovery would also find shadow APIs your team doesn't know about — deprecated endpoints, test environments, partner integrations."
- Show the traffic volume per endpoint if data has populated.

Part B: "Here's the schema we're enforcing"

Open **Schema Validation**. Show the uploaded schema. Talk through:

- "This is your OpenAPI spec — the contract that defines what valid requests look like. Notice the `POST /api/v1/cart/add` endpoint requires `product_id` as an integer and `quantity` between 1 and 10, and does not allow additional properties."

Part C: "Watch what happens to a valid request"

From a terminal visible to the audience:

```powershell
# Valid request — goes through fine
curl.exe -X POST https://peakcart-api.YOUR-SUBDOMAIN.workers.dev/api/v1/cart/add `
  -H "Content-Type: application/json" `
  -d "{\"product_id\": 2, \"quantity\": 3}"
```

> **PowerShell quoting note:** Use backtick (`` ` ``) for line continuation (not backslash). Use escaped double quotes (`\"`) inside the `-d` string, or use single quotes around the entire JSON and double quotes inside:
>
> ```powershell
> curl.exe -X POST https://peakcart-api.YOUR-SUBDOMAIN.workers.dev/api/v1/cart/add -H "Content-Type: applicatior
> ```
>
> Single-quote wrapping works in PowerShell 7+. On older PowerShell 5.1, you must escape: `-d "{\"product_id\": 2, \"quantity\": 3}"`

Expected: `201` response with success.

Part D: "Now watch an invalid request" (The Money Moment)

Switch schema validation from **Log** to **Block** in the dashboard. Then:

```powershell
# Invalid: wrong types, extra field, quantity out of range
curl.exe -v -X POST https://peakcart-api.YOUR-SUBDOMAIN.workers.dev/api/v1/cart/add `
  -H "Content-Type: application/json" `
  -d "{\"product_id\": \"abc\", \"quantity\": 999, \"coupon_code\": \"SQLI-ATTEMPT\"}"
```

Expected: The request is **blocked by Cloudflare before it reaches your Worker**. You'll get a `403` or Cloudflare block page.

Talk track: *"Notice what just happened. The attacker tried to send a string where an integer was expected, a quantity far outside the allowed range, and injected an extra field that isn't in the schema. Our Worker never even saw this request — Cloudflare rejected it at the edge. That's the positive security model. We don't need to know every possible attack string; we just need to know what a legitimate request looks like."*

Part E: "And the undocumented endpoint"

```powershell
curl.exe https://peakcart-api.YOUR-SUBDOMAIN.workers.dev/api/v1/admin/users
```

If you've configured a **fallthrough rule** (template available at Security > WAF > Templates: "Mitigate API requests to unidentified endpoints"), this request gets blocked too. Talk track: *"This endpoint isn't in your* *schema. Maybe it's a test endpoint someone forgot to decommission. With the fallthrough rule, any request to an undocumented endpoint is logged or blocked. Shadow APIs eliminated."*

Part F: Show Security Events

Navigate to **Security > Events** (or **Security > Analytics**). Filter by `Service = API Shield - Schema validation`. Show:

- The blocked requests with details on why they failed validation
- The specific fields that didn't match the schema

Step 5: Cleanup After the Demo

```powershell
npx wrangler delete peakcart-api
```

Remove the schema from API Shield and delete the endpoints from Endpoint Management if desired.

Quick Reference: Files You'll Need

File	Purpose
`wrangler.toml`	Worker configuration
`src\index.js`	The PeakCart API code
`peakcart-schema.yaml`	OpenAPI spec for API Shield
Test commands (above)	Pre-demo traffic generation + live demo requests

---

Windows-Specific Gotchas

1. **`curl` vs `curl.exe`:** PowerShell aliases `curl` to `Invoke-WebRequest`. Always use `curl.exe` to invoke the real curl. Alternatively, run `Remove-Alias curl -Force` at the start of your session.
2. **Line continuation:** PowerShell uses backtick (`` ` ``) not backslash (`\`). Make sure there's no space after the backtick.
3. **JSON in `-d` flag:** Three options depending on your PowerShell version:

- PowerShell 7+: `curl.exe -d '{"key": "value"}'` (single quotes work)
- PowerShell 5.1: `curl.exe -d "{\"key\": \"value\"}"` (escaped double quotes)
- Either version: Save the JSON to a file and use `curl.exe -d @body.json`

4. **Encoding:** The `Set-Content -Encoding UTF8` commands above ensure your YAML and JS files don't get BOM-prefixed, which could cause parsing issues.
5. **Windows Terminal recommendation:** Use Windows Terminal with a PowerShell 7 profile. The split-pane feature is great for the demo — dashboard commands on one side, curl output on the other.

Pro Tips for the Live Demo

1. **Pre-load all terminal tabs.** Have one tab with valid requests ready, one with invalid requests, and the Cloudflare dashboard already logged in and navigated to API Shield.
2. **Use `curl.exe -v` for the invalid requests** so the audience can see the HTTP response code and Cloudflare headers in real time.
3. **Have the Security Events page open in a second browser tab.** After sending the invalid request, switch to Events to show the block reason appearing in near real-time.
4. **Name your schema descriptively.** When you upload it, call it "PeakCart Product Catalog v1.0" — it looks more polished than "schema.yaml."
5. **Rehearse the toggle from Log to Block.** The demo's dramatic moment is flipping that switch and then showing the invalid request get rejected. Make sure you know exactly where the toggle is so you don't fumble during the presentation.
6. **Pre-stage your curl commands.** Save your demo commands in a `.ps1` script or as Windows Terminal snippets so you can paste them cleanly during the live demo without typos.