

# SCRAPING SYSTEM

CONFIGURATION-DRIVEN WEB SCRAPING

## User Manual

VERSION 1.0.0

# Contents


01 Getting started 3

1.1 What Scraping System is 3

1.2 Install and run the Admin app 3

1.3 First launch 3

1.4 Using the app 4

1.5 Connecting a client 5

1.6 Responsible use 5

02 Building configs 6

2.1 Find your selectors 6

2.2 Static vs. browser mode 6

2.3 Building the config in the Admin UI 7

2.4 Test, fix, save 9

2.5 When something fails 10

03 Unity client 11

3.1 Requirements 11

3.2 The client scripts 11

3.3 Create the config asset 11

3.4 Call it from your code 12

3.5 Try the demo scene 12

3.6 Common issues 13

04 HTTP API 14

4.1 Scraping requests 14

4.2 Managing saved configs 14

4.3 Responses 15

4.4 Caching and async jobs 15

CHAPTER 01

# Getting started

Install the desktop app, start the backend, and connect your first client.

## 1.1 What Scraping System is

Scraping System is a configuration-driven web scraping system. You describe a website once, where its data lives and what to extract, as a saved config. Any client (the desktop Admin UI, a Unity game, a script, a web app) can then ask for it by name and get back structured JSON. There is no per-website code.

This manual covers everything in one place:

- **Chapter 1, Getting started:** installing and running the Admin app, and connecting a client.
- **Chapter 2, Building configs:** turning "I want this data from this website" into a saved config.
- **Chapter 3, Unity client:** connecting a Unity project and calling the backend from C#.
- **Chapter 4, HTTP API:** calling the backend from web apps, scripts, and other clients.

## 1.2 Install and run the Admin app

The desktop app is the normal way to use the system. It starts the backend for you and gives you a visual tool for building and testing scraping configs. No terminal and no Node.js install are required.

1. Get the installer: `Scraper Admin Setup 1.0.0.exe`.
2. Double-click it.
3. When the installation is finished, launch **Scraper Admin** from the Start Menu.

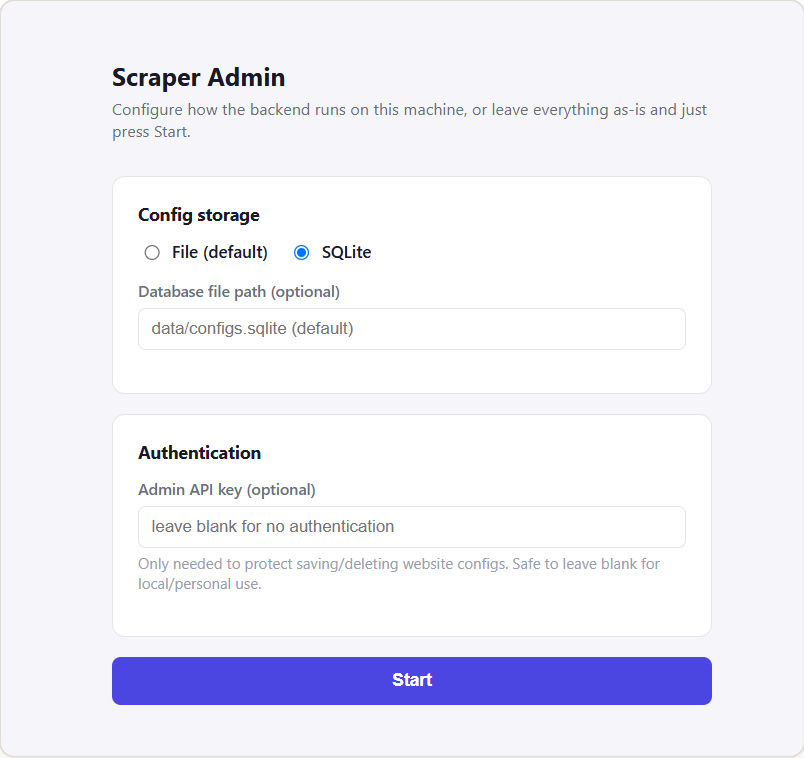
> **Windows SmartScreen.** Windows will likely show a blue "Windows protected your PC" warning. This is expected for an installer without a paid code-signing certificate, and it doesn't mean anything is wrong. Click **More info**, then **Run anyway**.

## 1.3 First launch

Every setting on the first screen is optional. The app works fully with zero configuration.

| Setting | What it does | Leave blank to... |
|---|---|---|
| **Config storage** | Where saved website configs live: plain JSON files (default) or a local SQLite database | Use JSON files in a `configs/` folder |
| **Database path** | Only used if you picked SQLite above | Use the default path ( `data/configs.sqlite` ) |
| **Admin API key** | A password required to create, update, or delete saved configs | Allow saving and deleting without a password (fine for personal or local use) |

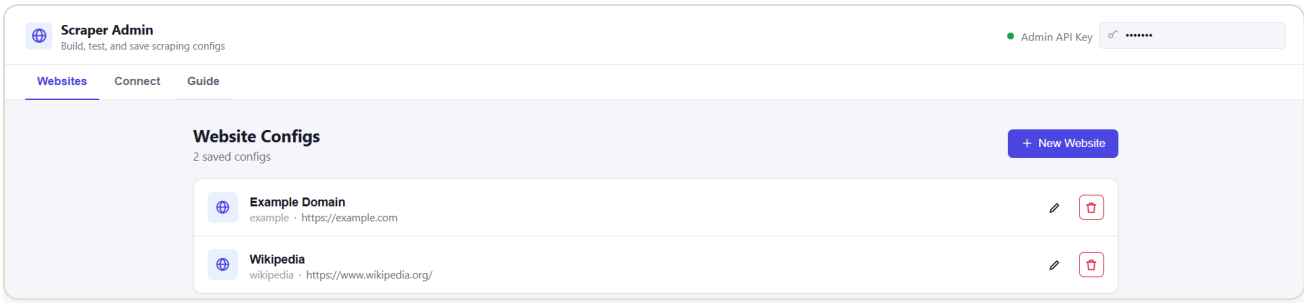
Click **Start**. The window switches to the Admin UI automatically once the backend is ready, which usually takes a couple of seconds.




## 1.4 Using the app

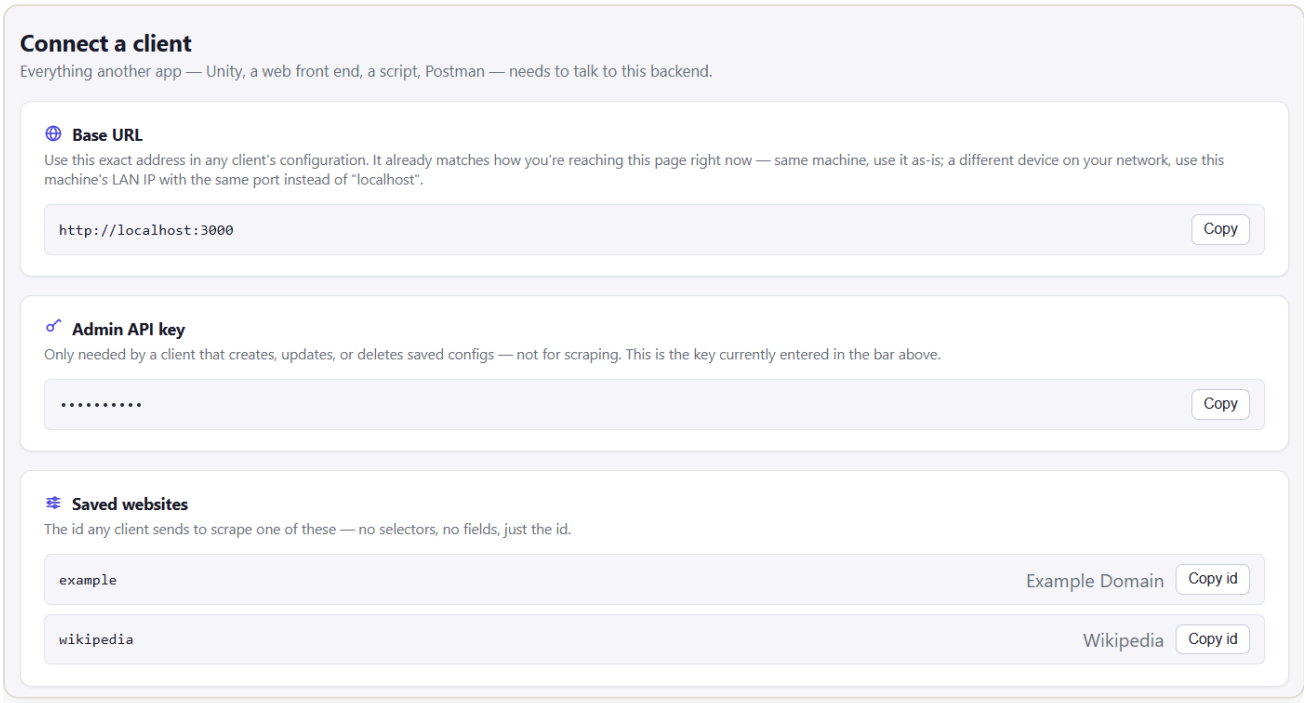
There are three tabs along the top:

- **Websites:** list, create, edit, test, and delete saved scraping configs. This is where you'll spend most of your time. Chapter 2, *Building configs*, explains exactly how to build a config for a real website.
- **Connect:** everything another app needs to talk to this backend: the exact URL to use, your admin key (if you set one), the id of every saved website, and copy-paste example requests in cURL, JavaScript, and C#/Unity. Use this when wiring up a new client.
- **Guide:** an in-app reference for every option in the config form, with worked examples.




## 1.5 Connecting a client

Open the **Connect** tab in the running app. It shows the exact URL and example code for your situation, generated live rather than as a static example.




For the full step-by-step Unity setup, see Chapter 3, *Unity client*.

## 1.6 Responsible use

This tool is intended for collecting data you are permitted to access. Respect each website's terms of service, robots policies, and copyright. Scraping System does not bypass CAPTCHAs, logins, or bot protection.

CHAPTER 02

# Building configs

How to turn “I want this data from this website” into a saved config in the Admin UI. No coding involved.

## 2.1 Find your selectors

This is the one step the app can't do for you. Before touching the Admin UI, open the target website in your own browser and figure out *where* the data you want lives in the page's HTML.

1. Right-click the piece of text, image, or link you want, then choose **Inspect**. This opens DevTools with that exact element highlighted.
2. Look at its `class` or `id` in the highlighted HTML. A class like `class="product-title"` becomes the selector `.product-title`, and an id like `id="price"` becomes `#price`.
3. **Test it before using it anywhere.** In the DevTools Console tab, type:

```
document.querySelectorAll(".product-title").length        // how many matches?
document.querySelectorAll(".product-title")[0].textContent // what's actually in it?
```

If the count looks wrong, the selector is wrong or too broad. Fix it here, before it ever reaches the Admin UI. This single habit will save you more time than anything else in this manual.

4. For a repeating list (every product, every article, every row), find the smallest element that wraps **one full item**. That's your item selector. Then find each field's selector **relative to that item**. It only needs to be unique *inside* one item, not on the whole page.
5. **Watch for duplicate or decoy matches.** Some sites reuse a class on more than one element per item, such as a hidden button or a second mobile-only copy of the same card. Always sanity-check the match *count* from step 3. If something matches 3 elements when you expected 1, narrow the selector by scoping it to a parent, for example `#results .load-more` instead of just `.load-more`.

## 2.2 Static vs. browser mode

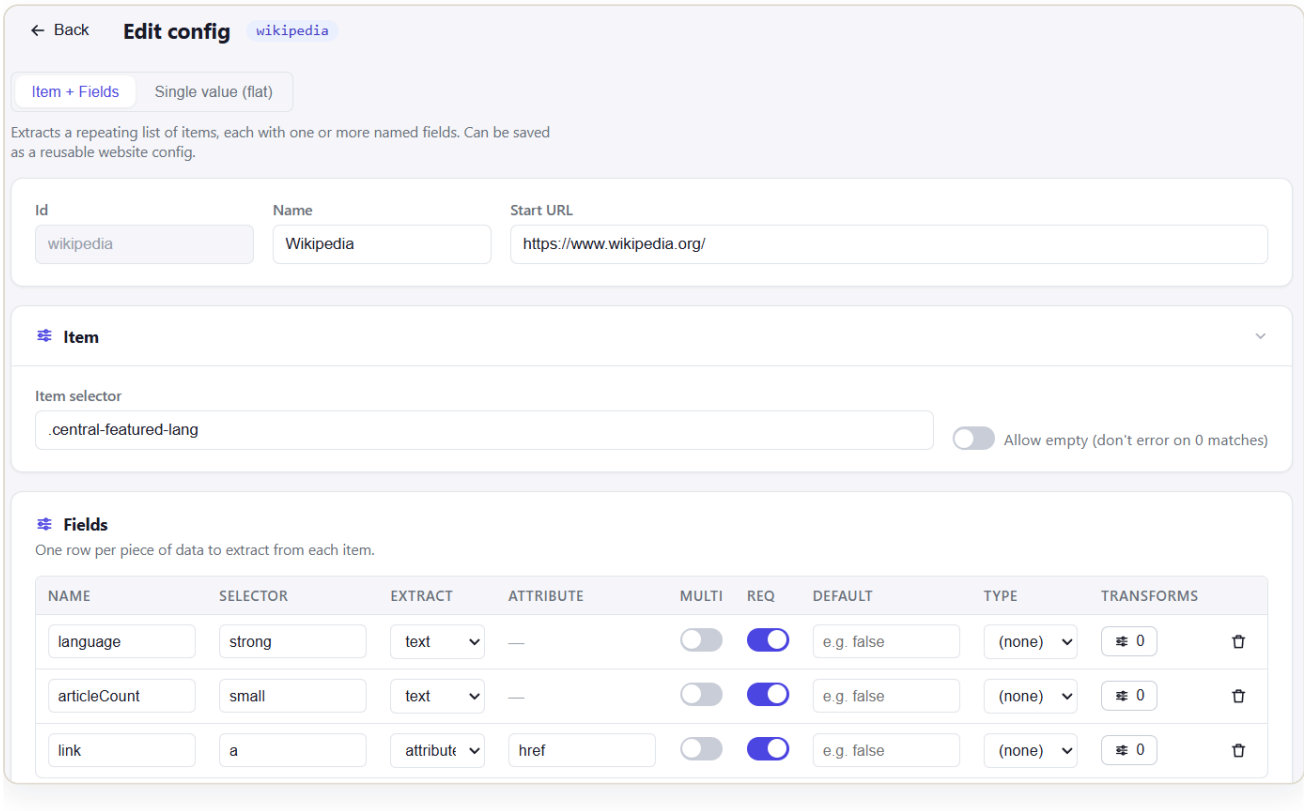
This decides whether your data even exists for the scraper to find, so get it right before anything else.

1. On the target page, right-click and choose **View Page Source**. Don't use Inspect for this: View Page Source shows the *raw* HTML before any JavaScript runs.
2. Press Ctrl+F and search for the text you want.

3. **Found it?** Use Scraper type `static`. It's the default, and much faster.
4. **Not found, but visible on the real page?** The site builds it with JavaScript after loading. Use Scraper type `browser`, which runs a real headless browser that executes the page's JavaScript before reading it.
5. **Still missing in browser mode?** The content probably loads after a delay, on scroll or after its own API call. Set **Wait for selector** to something that only exists once that content has loaded.

## 2.3 Building the config in the Admin UI

Open the **Websites** tab and click **+ New Website**.




### Mode: Item + Fields vs. Single value (flat)

- **Item + Fields** (the default) is for a repeating list: every product, every article. It requires an item selector and at least one field.
- **Single value (flat)** is for one value, or every match of one selector with no per-item wrapping. For example, "give me the whole page's HTML" (selector `html`, extract `html`) or "every link on the page" (selector `a`, extract `attribute`, attribute `href`). Flat mode can be tested but not saved as a reusable config. The backend only supports saving the Item + Fields shape.

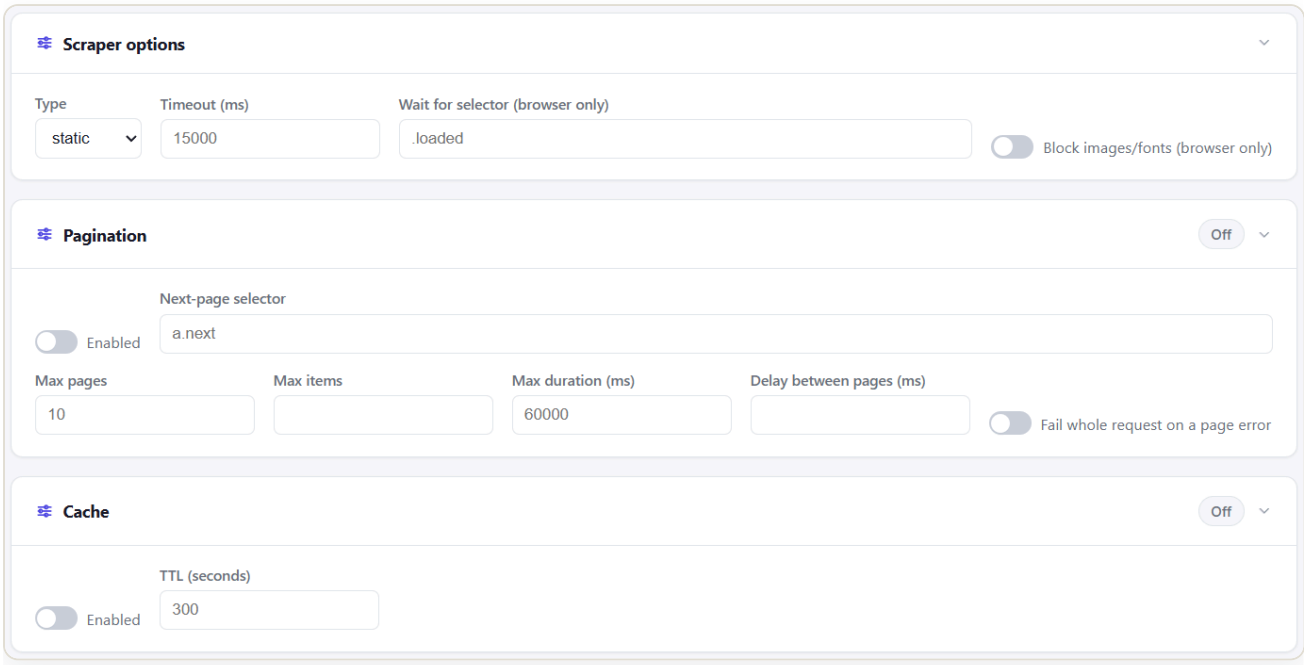
### Field options

| Option | Values | Default | What it does |
|---|---|---|---|
| Selector | any CSS selector | *(required)* | Where to find this value, relative to the item |
| Extract | `text` / `html` / `attribute` | *(required)* | `text` = visible text, `html` = inner markup, `attribute` = one HTML attribute (also set **Attribute**, e.g. `href` or `src`) |
| Multi | on/off | off | Off = first match only. On = every match, as a list (e.g. several tags on one product) |
| Req | on/off | on | On = fail the whole item if this selector isn't found. Off = quietly fall back to **Default** instead |
| Default | any value | — | Used only when Req is off and nothing matched. Type `false`, `0`, or plain text; it's parsed automatically |
| Type | `string` / `number` / `boolean` / `url` | — | A check, not a conversion: after transforms run, the value must already be this type or the scrape fails. Pair it with a transform that produces that type |
| Transforms | see below | — | A cleanup chain, run in order |

### Transforms

| Name | What it does |
|---|---|
| `trim` | Strips leading and trailing whitespace |
| `removeCurrency` | Strips `$`, `€`, commas, etc. from a price-like string |
| `parseNumber` | Converts a numeric string to a real number (has a decimal-separator option for "1.299,00"-style numbers) |
| `absoluteUrl` | Turns a relative link ( `/item/42` ) into a full URL |
| `regex` | Extracts a regex match or capture group (pattern, optional group number, optional flags) |
| `replace` | Regex find-and-replace (pattern, replacement, optional flags) |
| `default` | Substitutes a fixed value when the extracted value is empty. It's the only transform that still runs on an empty value |

For example, the chain `trim` → `removeCurrency` → `parseNumber` turns `" $1,299.00 "` into the number `1299`.

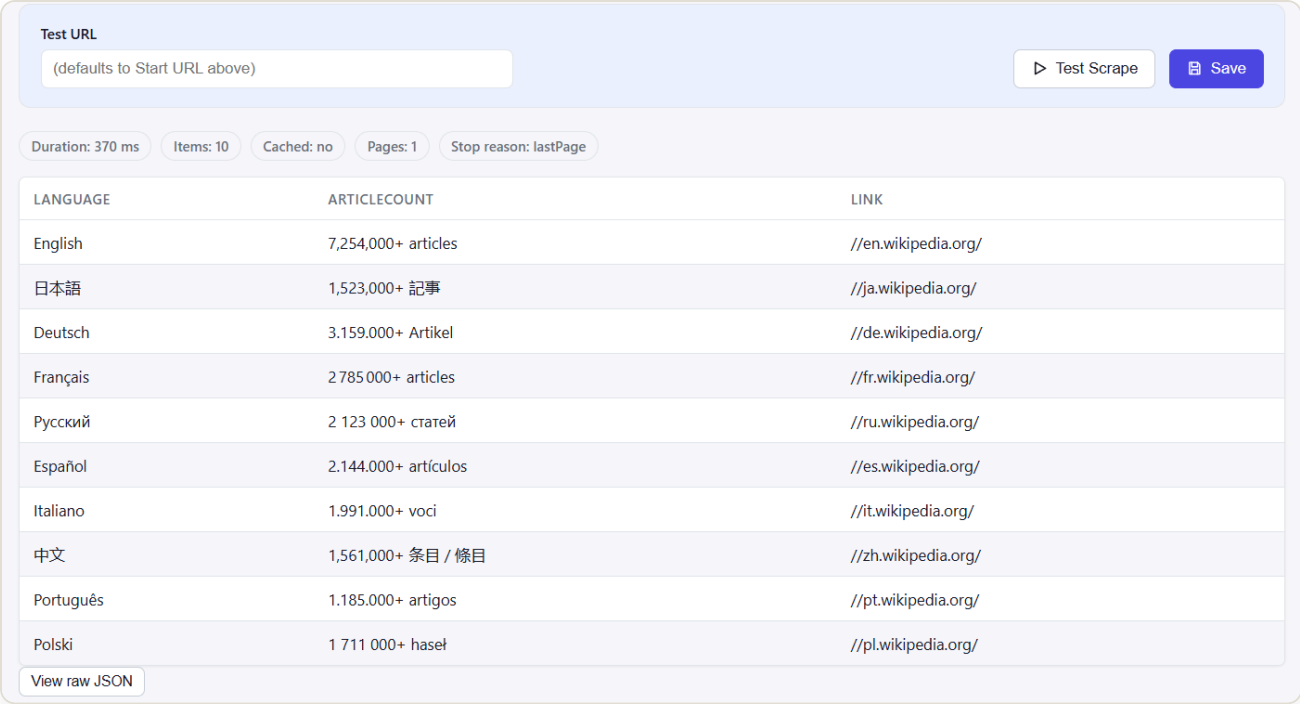



### Scraper, Pagination, and Cache

- **Scraper: Type** is `static` or `browser`, as decided above. **Timeout** is how long to wait before giving up. **Wait for selector** works in browser mode only. **Block images/fonts** also works in browser mode only, and is on by default for speed.
- **Pagination:** turn it on and set a **Next-page selector**, the "next page" link. Double-check that it isn't matching a decoy elsewhere on the page. **Max pages**, **Max items**, and **Max duration** cap how far it goes. The result tells you exactly why it stopped: it ran out of pages ( `lastPage` ), hit a limit ( `maxPages` , `maxItems` , or `timeout` ), or a later page failed ( `error` ).
- **Cache:** turn it on with a TTL in seconds to reuse a recent identical result instead of re-scraping. The results panel shows a `Cached: yes` badge when this kicks in.

## 2.4 Test, fix, save

1. Click **Test Scrape**. The results table shows exactly what came back. If something's wrong, an error banner shows the exact code and message, telling you which field or selector failed, not just "it broke."
2. Fix your selectors based on what you see, and repeat.
3. Once it looks right, click **Save**. The config is now reusable by id from any client. The **Connect** tab has copy-paste examples in cURL, JavaScript, and Unity/C#.




## 2.5 When something fails

| Error code | What it means |
|---|---|
| `SELECTOR_NOT_FOUND` | That selector matched nothing on the page (or inside the item). Re-check it in DevTools. |
| `CONFIG_NOT_FOUND` | The website id you asked for doesn't exist. Check the Websites list. |
| `URL_NOT_ALLOWED` | The target URL resolves to a private or internal address, or uses a disallowed protocol. It's blocked for safety; this isn't a selector problem. |
| `TIMEOUT` | The page took too long. Raise **Scraper → Timeout**, or try browser mode if the site renders slowly. |
| `INVALID_CONFIGURATION` | The request doesn't match the expected shape, for example an empty string where a real URL or value was required. Check for blank optional fields that should either have a real value or be left out entirely. |

A site that actively blocks automated requests (bot detection, CAPTCHA challenges) will fail in ways no selector fix can solve. That's the target site's own defense working as intended, not a bug in a config.

CHAPTER 03

# Unity client

Connect a Unity project to the backend. Unity never needs to know about selectors, CSS, or page structure: it asks for a saved website by id and gets data back.

## 3.1 Requirements

- Unity 6000.x or later. The client was built and verified against 6000.3.10f1.
- The **Newtonsoft Json.NET** package. Add it in the Package Manager: **Window → Package Manager → + → Add package by name...**, then enter `com.unity.nuget.newtonsoft-json`.
- The Scraper Admin desktop app running, on the same machine or on your LAN (see Chapter 1).

## 3.2 The client scripts

After importing the package, the client is organized like this:

```
Networking/Scraper/
    ScraperClient.cs                the class your game code calls
    ScraperClientConfig.cs          a ScriptableObject holding the backend URL and key
    Models/                         result types, error types, generic record type
    Internal/                       HTTP and JSON plumbing (not called directly)
    UI/ScraperFeedbackPanel.cs      optional: a visual status/log HUD panel for a Canvas
Demo/
    ExampleItem.cs, ScraperDemoController.cs   optional: a working example scene script
```

You don't need to modify any of these files. Only `ScraperClientConfig`, a data asset rather than code, needs your input.

## 3.3 Create the config asset

In the Project window, right-click and choose **Create → Scraper → Client Config**. Select the new asset and fill in:

| Field | Value |
|---|---|
| **Base Url** | The URL from the Admin app's **Connect** tab ("Base URL" box, with a Copy button). Usually `http://localhost:3000` if Unity and the backend run on the same machine. |

| Field | Value |
| --- | --- |
| **Request Timeout Seconds** | Leave at the default (15) unless you have a reason to change it. |
| **Admin Api Key** | Only needed for the admin methods ( `CreateWebsite` , `UpdateWebsite` , `DeleteWebsite` ); also copyable from the Connect tab. Leave blank for normal gameplay scraping. |

> **Different machines?** If the backend runs on a different computer than Unity, use that machine's LAN IP instead of `localhost` (for example `http://192.168.1.50:3000` ), and make sure Windows Firewall on that machine allows inbound connections on port 3000.

## 3.4 Call it from your code

Drag the config asset into a `[SerializeField] ScraperClientConfig` reference, then:

```
var client = new ScraperClient(myConfig);

// Simplest path: no C# class needed per website, read fields by name
var result = await client.ScrapeWebsiteRaw("your-website-id");
if (result.IsSuccess)
{
    foreach (var item in result.Value)
    {
        string title = item["title"]?.ToString();
    }
}
else
{
    Debug.LogError($"{result.Error.Code}: {result.Error.Message}");
}
```

## 3.5 Try the demo scene

Before writing your own code, see the client working:

1. Attach `ScraperDemoController` (from the `Demo/` folder) to an empty GameObject.
2. Assign your `ScraperClientConfig` .
3. Press Play.

It runs through six request paths (typed, raw, an expected error, async, and the fully generic ScriptableObject-driven path) and logs each result to the Console.

Optionally, assign a `ScraperFeedbackPanel` to its **Feedback** slot to see the same progress as a status and log HUD at runtime: create an empty GameObject under a Canvas and add that component.




## 3.6 Common issues

| Symptom | Fix |
|---|---|
| Every call fails with a network error | The backend isn't running. Open the Scraper Admin app first. |
| Works in the Unity **Editor** but not in a **standalone build** | In **Player Settings → Player → Other Settings**, set **Allow downloads over HTTP** to `Always allowed`. The backend uses plain `http://`, not `https://`, and Unity blocks that by default in built players. |
| `ConfigNotFound` error | The website id doesn't exist on this backend. Check the **Websites** tab (or `GET /websites`) for the exact id. |
| A `ScraperWebsiteDefinition`'s **Url Override** field causes an `INVALID_CONFIGURATION` / "Invalid URL" error | Leave it completely blank, not a placeholder string. A blank Inspector field is sent as `""`, and an explicit empty string there is invalid. (Already fixed in the current client; mentioned in case you're on an older copy.) |

CHAPTER 04

# HTTP API

For clients other than the included Unity client: web apps, scripts, tools, and other backends. Any language that can send an HTTP request can use it.

## 4.1 Scraping requests

Send requests to the Base URL shown in the app's **Connect** tab (usually `http://localhost:3000`). `POST /scrape` accepts one of three request shapes:

```
// 1. Saved config by id (the normal way)
{ "website": "example" }

// 2. Inline config: item + fields, no saved config needed
{ "url": "https://example.com", "config": { "item": {...}, "fields": {...} } }

// 3. Flat: one field, every match
{ "url": "https://example.com", "selector": "title", "extract": "text" }
```

For example, from a terminal:

```
curl -X POST http://localhost:3000/scrape \
  -H "Content-Type: application/json" \
  -d '{"website": "example"}'
```

## 4.2 Managing saved configs

```
GET    /websites           list saved configs
GET    /websites/:id       fetch one
POST   /websites           create               }
PUT    /websites/:id       replace (or create)  }  need "Authorization: Bearer <admin key>"
DELETE /websites/:id       remove               }  when an admin key is set
```

The admin key is the one you entered on the app's first launch screen; the **Connect** tab shows it with a Copy button. If you didn't set one, these routes are open.

**Endpoint reference**

| METHOD & PATH | NEEDS THE ADMIN KEY? | WHAT IT DOES |
|---|---|---|
| `POST /scrape` | No | Run a saved config — `{"website":"<id>"}` |
| `GET /websites` | No | List saved configs |
| `GET /websites/:id` | No | Fetch one full saved config |
| `POST /websites` | Yes* | Create a new saved config |
| `PUT /websites/:id` | Yes* | Create or replace a saved config |
| `DELETE /websites/:id` | Yes* | Remove a saved config |
| `GET /jobs/:id` | No | Poll an async ("async":true) scrape job |

*Only if this backend has `ADMIN_API_KEY` set at all — otherwise these are open too (with a startup warning logged on the server).

## 4.3 Responses

Every response has the same shape:

- On success: `{ success, url?, data?, metadata? }`
- On failure: `{ success: false, error: { code, message } }`

`metadata` describes the run, for example the duration, the number of items and pages, whether the result came from the cache, and why pagination stopped. The error codes are listed in Chapter 2, *When something fails*.

The **Connect** tab in the running app has live, copy-paste examples in cURL, JavaScript, and C#.

## 4.4 Caching and async jobs

Add `"cache": { "enabled": true, "ttl": 300 }` to any request (or to a saved website config) to cache its result in memory for that many seconds. Repeat identical requests return instantly with `metadata.cached: true` instead of re-scraping. The cache is kept in memory, so it's cleared when the app closes.

Add `"async": true` to any request to get an instant `202 { jobId, status: "queued" }` instead of waiting. This is useful for slow `scraper.type: "browser"` scrapes. Poll `GET /jobs/:jobId` until `status` is `"completed"` (with `result`) or `"failed"` (with `error`).