Lyftr AI — Full-stack Assignment

Universal Website Scraper (MVP) + JSON Viewer

Resources policy: You may use any resources — including coding assistants (Copilot, Cursor, ChatGPT, etc.). There is **no extra credit** for avoiding AI; we care about your reasoning and final result.

---

0. Overview

You will build a **universal website scraper** (MVP) and a small **JSON viewer frontend**.

Given a URL, your system should:

1. Scrape the page (handle **static** and **JS-rendered** content).
2. Perform a basic **click flow** (tabs or "Load more/Show more").
3. Support **scroll/pagination to depth ≥ 3** (i.e., at least 3 loads/pages).
4. Return **section-aware JSON** following the provided schema.
5. Provide a minimal frontend to input a URL and view/download the JSON.

The assignment is designed so we can run all submissions in a uniform way and evaluate them consistently.

---

1. Tech Stack & Run Instructions

1.1. Required stack

- **Language:** Python 3.10+
- **Backend:** FastAPI (preferred) or Flask
- **HTML rendering:**

    - Static: `httpx` or `requests` + `selectolax` / `beautifulsoup4` / `lxml`
    - JS: **Playwright (Python)**
- **Frontend:**
    - Either: Jinja2 template rendered by the backend
    - Or: a minimal SPA (e.g., React/Vite) calling your backend API
- **Server runtime:** `uvicorn` or `gunicorn` (or Flask's dev server is acceptable for this assignment)

1.2. How your project must run

Your repo **must** contain:

- A **shell script**: `run.sh`

When we execute:

chmod +x run.sh

./run.sh

it must:

1. Create/activate a virtual environment (if needed).
2. Install dependencies (e.g., `pip install -r requirements.txt` or `pip install -e .`).
3. Start your web server on: **http://localhost:8000**

We will **not** use Docker to run your project.
You may include Docker files if you like, but evaluation will use `./run.sh`.

---

2. API Specification

We will interact with your backend via:

- `GET /healthz`
- `POST /scrape`

2.1. `GET /healthz`

- **Method:** `GET`
- **Request body:** none
- **Response body (minimum):**

{ "status": "ok" }

You may include additional fields, but `"status": "ok"` must be present when the server is healthy.

2.2. `POST /scrape` — Request

- **URL:** `/scrape`
- **Method:** `POST`
- **Body (JSON):**

{

"url": "https://example.com"

}

Rules:

- `url` must be an **http(s)** URL.

- Non-http(s) schemes (e.g., `file://`) should be rejected or handled gracefully with a clear error.

2.3. `POST /scrape` — Response

Your response must be JSON of the form:


```
{
 "result": {
  "url": "https://example.com",
  "scrapedAt": "2025-11-13T00:00:00Z",
  "meta": {
   "title": "Page Title",
   "description": "Meta description",
   "language": "en",
   "canonical": "https://example.com"
  },
  "sections": [
   {
    "id": "hero-0",
    "type": "hero",
    "label": "Hero",
    "sourceUrl": "https://example.com",
    "content": {
     "headings": ["Welcome"],
     "text": "Short summary…",
     "links": [
      { "text": "Get Started", "href": "https://example.com/get-started" }
```

```
      ],
      "images": [
        { "src": "https://example.com/img.png", "alt": "Hero" }
      ],
      "lists": [
        ["Item 1", "Item 2"]
      ],
      "tables": []
    },
    "rawHtml": "<section>…</section>",
    "truncated": true
  }
],
"interactions": {
  "clicks": [
    "button[aria-controls='features']",
    "button:contains('Load more')"
  ],
  "scrolls": 3,
  "pages": [
    "https://example.com",
    "https://example.com/?page=2",
    "https://example.com/?page=3"
  ]
},
"errors": [
  { "message": "Timeout waiting for #main", "phase": "render" }
```

```
  ]
 }
}
```

Required fields inside `result`

We will expect the following keys to exist with these types:

- `url`: string — must exactly match the input URL.
- `scrapedAt`: string — ISO8601 datetime (UTC or with timezone).
- `meta` (object):
    - `title`: string (may be empty).
    - `description`: string (may be empty).
    - `language`: string (e.g., `"en"`; can be a best guess).
    - `canonical`: string URL or `null`.
- `sections`: **non-empty array** of objects, each with:
    - `id`: string — a stable identifier for this section.
    - `type`: one of `hero | section | nav | footer | list | grid | faq | pricing | unknown`
    - `label`: human-readable label for the section (e.g., `"Hero"`, `"Features"`).
        - If there is no explicit heading, derive a label from the first 5–7 words of the text.
    - `sourceUrl`: the URL that this section's HTML came from.
    - `content` (object):

        - `headings`: array of strings.
        - `text`: string containing the main text for the section.
        - `links`: array of `{ "text": string, "href": string }` with **absolute URLs**.
        - `images`: array of `{ "src": string, "alt": string }`.
        - `lists`: array of arrays of strings (e.g., each inner array is a list).
        - `tables`: array (you can choose the shape; we only require it to be an array).
    - `rawHtml`: string — truncated HTML snippet representing the section.
    - `truncated`: boolean — `true` if you truncated `rawHtml`, `false` otherwise.
- `interactions` (object):
    - `clicks`: array of strings — CSS selectors or short descriptions of the elements you attempted to click.
    - `scrolls`: integer — number of scroll actions performed.
    - `pages`: array of strings — list of visited page URLs (absolute).
- `errors`: array of objects of the form:
    - `{ "message": string, "phase": string }`
      (Use `"phase"` values such as `"fetch"`, `"render"`, `"parse"`, etc. as you see fit.)

You may add more fields if you find them useful, but the above fields must be present and correctly typed.

---

3. Backend Requirements

3.1. Static scraping

Implement static scraping using `httpx` or `requests` plus an HTML parser (`selectolax`, `beautifulsoup4`, or `lxml`):

- Fetch the raw HTML for the input URL.
- Extract:
  - `meta.title` from `<title>` or `<meta property="og:title">`.
  - `meta.description` from `<meta name="description">` or similar.
  - `meta.language` from `<html lang="…">` or a best-effort guess.
  - `meta.canonical` from `<link rel="canonical">` or `null` if missing.
- Group content into `sections` using:
  - Landmarks (`header`, `nav`, `main`, `section`, `footer`) and/or
  - Headings (`h1`–`h3`) and their following content.

For each section:

- Populate `content.headings`, `content.text`, `content.links`, `content.images`, `content.lists`, `content.tables`.
- Generate a fallback `label` if needed using the first 5–7 words of the section text.
- Make `links.href` absolute URLs based on the page URL.
- Set `rawHtml` to a truncated HTML snippet:
  - You may limit this by character count.
  - Set `truncated` appropriately.

3.2. JS rendering & fallback

Implement a **fallback strategy**:

- Attempt **static** scraping first.

- If static HTML appears insufficient (e.g., missing main content, too little text, or based on a heuristic you design), fall back to **Playwright**.
- When using Playwright:
    - Launch a browser context.
    - Navigate to the URL.
    - Wait appropriately (for network idle, key selectors, or a sensible combination).
    - Extract HTML and reuse your existing section parsing logic.

You should document your heuristic and wait strategy in `design_notes.md`.

3.3. Click flows & scroll/pagination

Implement behaviour to explore content beyond the initial viewport:

- At least **one** of:
    - Clicking tabs (e.g., `[role="tab"]`, buttons that switch content panes).
    - Clicking “Load more/Show more” style buttons.
- Implement **scrolling and/or pagination** to reach a depth of **at least 3**:
    - Infinite scroll: scroll down and wait for new content to load, at least 3 times; or
    - Pagination links: follow “next page” links up to at least 3 pages; or
    - A combination of the two.

Record these interactions in the `interactions` object:

- `clicks`: attempted click selectors or labels.
- `scrolls`: number of scroll operations.
- `pages`: distinct URLs visited during the scrape.

3.4. Noise filtering, errors, and limits

- Implement basic noise filtering:
    - Avoid or strip obvious overlays like cookie banners, modals, or newsletter popups when possible (e.g., via CSS selectors).
- Implement reasonable timeouts to avoid hanging indefinitely.
- When errors occur (timeouts, blocked by automation, invalid URL, etc.):
    - Populate `errors[]` with appropriate `message` and `phase`.
    - Return the partial data you have, if it is safe and consistent, rather than crashing.

---

4. Frontend Requirements

Serve a simple UI at the root path: `GET /`

The frontend can be:

- A Jinja2 HTML template rendered by your backend, or
- A small SPA (e.g., React/Vite) bundled and served by your backend.

The UI should:

1. Provide an **input box** for `url`.
2. Provide a **"Scrape"** button to submit the URL to `POST /scrape`.
3. Show a **loading state** while the request is in progress.
4. Show any **error messages** returned from the backend.
5. Render the parsed `sections` as:
    - A list or accordion of sections.
    - Each entry showing at least `label` (and optionally `type`).

    - The ability to expand a section and see its JSON (`content`, `rawHtml`, etc.) in a readable format.

6. Offer a way to **download** the full `result` JSON (e.g., a "Download JSON" button).

Design can be minimal; clarity and usability are more important than visuals.

---

5. Required Files

Your repository **must** include these files at the root:

1. `run.sh`
2. `requirements.txt` (or `pyproject.toml`/`Pipfile`, with matching instructions in README)
3. `README.md`
4. `design_notes.md`
5. `capabilities.json`

5.1. `README.md`

Please include:

- How to set up and run the project, including:

  chmod +x run.sh

  ./run.sh

- Any environment details (e.g., "`run.sh` runs `playwright install`").
- The **three primary URLs** you used for testing, with a short note about each, e.g.:

  - https://en.wikipedia.org/wiki/Artificial_intelligence — static page

  - https://vercel.com/ — JS-heavy marketing page with tabs

  - https://news.ycombinator.com/ — pagination to depth 3

- Any known limitations or caveats.

5.2. `design_notes.md`

Use the following structure (you can add more detail, but keep these headings):

Design Notes

Static vs JS Fallback

- Strategy: [describe when/how you decide to use JS rendering vs static]

Wait Strategy for JS

- [ ] Network idle

- [ ] Fixed sleep

- [ ] Wait for selectors

- Details: [1–3 sentences describing what you actually did]

Click & Scroll Strategy

- Click flows implemented (e.g., tab click, load more):

- Scroll / pagination approach:

- Stop conditions (max depth / timeout):

Section Grouping & Labels

- How you group DOM into sections:

- How you derive section `type` and `label`:

Noise Filtering & Truncation

- What you filter out (e.g., cookie banners, overlays):

- How you truncate `rawHtml` and set `truncated`:

5.3. `capabilities.json`

Fill this with booleans that describe what you believe you implemented, for example:

```
{
  "static_scraping": true,
  "js_rendering": true,
  "click_tabs": false,
  "load_more_clicks": true,
  "infinite_scroll": true,
  "pagination_links": true,
  "noise_filtering": true,
  "html_truncation": true
}
```

Be honest — we use this together with automated tests.

---

6. Suggested Test URLs (for you)

You may choose different sites, but it helps to pick from these or similar:

Static / Largely Static

- Wikipedia — `https://en.wikipedia.org/wiki/Artificial_intelligence`
- MDN — `https://developer.mozilla.org/en-US/docs/Web/JavaScript`

JS-Rendered / Tabs

- Vercel — `https://vercel.com/`

- MUI Tabs — `https://mui.com/material-ui/react-tabs/`
- Next.js Docs — `https://nextjs.org/docs`

Pagination / "Load more" / Infinite Scroll

- Hacker News — `https://news.ycombinator.com/`
- Dev.to — `https://dev.to/t/javascript`
- Unsplash — `https://unsplash.com/s/photos/nature`
- Infinite Scroll demo — `https://infinite-scroll.com/demo/full-page/`

---

7. Constraints & Notes

- Only **http(s)** URLs are supported.
    - Reject or safely fail for `file://` and other schemes.
- You may limit scraping to the same origin (single domain) for simplicity.
- Add sensible timeouts and avoid aggressive retries.
- If a site blocks automation or is not compatible, return a clear error in `errors[]` and mention it in your README.

---

8. Testing & Evaluation

This section describes how we will evaluate your submission.
You do **not** need to implement anything extra beyond the requirements above, but this is how we'll interpret and test your work.

We will evaluate in **5 stages**:

Stage 1 — Server & Health Check

- Run `./run.sh` and expect your server on `http://localhost:8000`.
- Call `GET /healthz` and expect:
    - Valid JSON.
    - `"status": "ok"`.

If this fails, later stages may not be evaluated.

Stage 2 — Static Scraping & Basic JSON

- Call `POST /scrape` with one or more primarily static URLs.
- Check:
    - Response parses as JSON.
    - The `result` object exists with all required fields.
    - `result.url` matches the input URL.
    - `sections` is a non-empty array.
    - At least one section has non-empty `content.text`.
    - `links.href` are absolute URLs.

Stage 3 — JS Rendering & Fallback

- Call `POST /scrape` on one or more JS-heavy URLs.
- Check that:
    - Content that is only visible after JS execution appears in the `sections` output.
    - Static-first then JS-fallback behaviour appears to be in effect (for example, via documented strategy in `design_notes.md` and richer output versus what a

static fetch would provide).

If you optionally include a `meta.strategy` field (e.g., `"static"` vs `"js"`), we may also use that as a signal.

Stage 4 — Click Flows & Scroll/Pagination Depth ≥ 3

- Call `POST /scrape` on URLs that require interaction (tabs / "Load more" / pagination / infinite scroll).
- Check that:
    - `interactions.clicks` contains at least one selector/description for a meaningful click.
    - `interactions.scrolls` is ≥ 2 when scrolling is needed.
    - `interactions.pages` contains at least 3 visited page URLs for an interaction-heavy page (depth ≥ 3).

We may inspect `sections` to ensure that more content is indeed being loaded versus the initial page.

Stage 5 — Frontend JSON Viewer

- Open `http://localhost:8000/` in a browser.
- Manually verify that:
    - We can enter a URL and trigger a scrape.
    - Some representation of `sections` appears (e.g., list/accordion).
    - Expanding a section shows structured data.
    - There is a way to download the full JSON (or at least view it in a consolidated way).

This stage focuses on basic usability and integration with your backend.

Scoring (high-level)

While the exact scoring may change, broadly:

- **Core functionality (Stages 1–3)**: ability to run, static + JS scraping, and correct JSON shape.
- **Depth & robustness (Stage 4)**: interaction handling and navigation depth.
- **Usability & clarity (Stage 5 + docs)**: frontend, README, `design_notes.md`, and honest `capabilities.json`.

---

9. Submission

- Share a **GitHub repository link**.
- Email to **careers@lyftr.ai** with subject:
  `Full-Stack Assignment — [Your Name]`
- In the email or README, list the three primary URLs you used to test your scraper.