HTML Cleaning for Scraping

Is there a potent code or library out there that cleans loaded html content "of dynamic javascript web pages using the ScrapingFish tool". I want to strip it from all the unnecessary data (that makes it very difficult for an LLM or a human to read the relevant data for web scraping and data ingestion purposes). Please provide the code for such task.

To give you the most effective solution, could you please clarify a few details?

1. What kind of data are you trying to retain from the HTML (e.g., main article content, tables, product listings)?
2. Do you want to keep HTML structure (like <p>, <h1>, <table>) or just extract clean text?
3. What programming language are you using (e.g., Python)?
4. Do you have specific libraries you prefer to use (besides ScrapingFish)?
5. Do you need the solution to work with multiple languages or just English content?

Once I have this info, I can generate a powerful and efficient code solution for your use case.

What kind of data are you trying to retain from the HTML (e.g., main article content, tables, product listings)? firm's and companies' professional business data, contact info, service description, etc.

Do you want to keep HTML structure (like <p>, <h1>, <table>) or just extract clean text? you can keep html structure

What programming language are you using (e.g., Python)? python

Do you have specific libraries you prefer to use (besides ScrapingFish)? not really

Do you need the solution to work with multiple languages or just English content? just English

Thanks! I'll put together a Python-based solution that leverages ScrapingFish for rendering JavaScript-heavy pages, then intelligently cleans the returned HTML to extract only relevant

company-related data—like business name, contact info, and service descriptions—while retaining useful HTML structure. I'll also identify the best library or combination of libraries for this task.

I'll let you know as soon as I have the results.

Clean HTML Output from ScrapingFish for Business Data Extraction

Scraping and Cleaning Dynamic Web Page Content using Python and ScrapingFish

Introduction and Problem Overview

Dynamic web pages that rely on JavaScript often require special handling to scrape, since a simple HTTP request may not capture content rendered client-side. In this task, we need to **load a web page with fully rendered HTML (including JavaScript content)** and then clean it to retain only relevant business-related information – such as company names, contact details, service descriptions, etc. We must strip away extraneous elements like advertisements, navigation menus, footers, scripts, and other non-informational sections. The end result should be either cleaned HTML (preserving basic structure like headings, paragraphs, tables) or structured text that contains **only the relevant content**.

Key challenges:

- Loading JavaScript-driven content: requiring a tool or API that can render the page (since libraries like `requests` alone won't execute JS).
- Parsing and cleaning the HTML while focusing on **English content** and removing boilerplate (repeated template elements).
- Preserving useful structure (e.g. `<h1>`, `<p>`, `<div>`, `<table>`) for readability and context, rather than dumping plain text.

Solution approach in brief:

1. **Fetch the dynamic page's HTML** using the ScrapingFish API (a scraping proxy that uses real browsers to render JS content).
2. **Parse the HTML** in Python (e.g. with Beautiful Soup and/or lxml) to obtain a DOM tree of the page.
3. **Remove non-informational elements** by targetting specific tags and sections: e.g. drop all `<script>` and `<style>` tags, navigation bars, headers/footers, ads, and any other clutter.
4. **Extract the main content** – focusing on text-heavy elements (paragraphs, headings, lists, tables) that contain business information. Use a boilerplate removal library or algorithm to filter out leftover navigation links or repetitive text. Ensure only English text content remains (skip other languages if present).
5. **Output the cleaned content** either as HTML (containing just the relevant sections in a simplified structure) or as structured text (e.g. plain text paragraphs, possibly with markdown or other notation if needed).

The sections below discuss the tools and libraries to achieve this, followed by a complete Python code example and rationale for the chosen approach.

Key Libraries and Tools for HTML Retrieval and Cleaning

To solve this problem, we'll leverage a combination of open-source libraries and services, each responsible for a piece of the workflow:

- **ScrapingFish API + Requests** – for loading pages with JavaScript. ScrapingFish is a web scraping API that uses real browsers under the hood: you provide your API key and target URL, and it returns the fully rendered HTML . By default, **JavaScript rendering is off**, so we enable it with a parameter (`render_js=true`) to ensure dynamic content is loaded . Using the Python `requests` library, we can call ScrapingFish's endpoint with our API key and URL to get the page's HTML content . (ScrapingFish handles rotating proxies and headless browsers for us.)

- **Beautiful Soup 4 (BS4)** – a popular Python library for parsing HTML and XML. BeautifulSoup makes it easy to load HTML into a parse tree and navigate or modify it with Pythonic syntax . We will use BS4 (with an HTML parser like *lxml*) to traverse the HTML DOM and remove unwanted nodes. It's very handy for tasks like finding all `<script>` tags and decomposing them, stripping out `<footer>` sections, etc., saving us a lot of manual string processing .
- **lxml** – a high-performance HTML/XML parsing library. BS4 can use *lxml* as its parser for speed and robustness. Additionally, **lxml** provides powerful XPath/CSS selectors and even an `lxml.html.Cleaner` utility that can drop scripts, style, and other unwanted elements in one go. We'll rely on lxml (via BS4 or directly) to efficiently handle the HTML structure.
- **jusText** – a specialized boilerplate removal tool. *jusText* is designed specifically to remove boilerplate content (navigation links, headers, footers, etc.) from HTML pages while **preserving the main textual content** . It uses heuristics (like looking for full sentences and language-specific stopwords) to identify informative paragraphs versus clutter. This makes it well-suited to isolating the meaningful text (e.g. descriptions, contact info) from common page furniture. We can use jusText with an English stopword list to focus on English content and filter out irrelevant fragments.
- **Trafilatura** (optional alternative) – an advanced text-extraction library that can handle crawling, downloading, and extracting main content. Trafilatura excels at turning *raw HTML into structured, meaningful data*, including the main text and metadata, while *avoiding the noise caused by recurring elements like headers and footers* . It integrates algorithms from jusText and Readability, striving for a balance between excluding clutter and retaining all important parts. Trafilatura could be used to directly get a cleaned HTML or text output (it even allows an `output_format="html"`). In our context, it's another powerful open-source option for boilerplate removal if we want an all-in-one solution.
- **Readability (readability-lxml)** (alternative) – an implementation of the Arc90 Readability algorithm in Python. Given an HTML document, Readability will extract and clean up the main body content and title . This library is effective for article-like pages and can output a simplified HTML of the primary content. It might be slightly less suited if the page's relevant info is more scattered (like a list of business details), but it's worth noting as a potent tool for content extraction.

- *(Other libraries worth mentioning include **Newspaper3k** and **Goose** for article extraction, or **html2text** and **inscriptis** for converting HTML to plain text. However, for our business content use case, the libraries above are the most directly applicable.)*

Each of these tools contributes to different stages: ScrapingFish+requests handles retrieval; BS4/lxml handles parsing and fine-grained cleaning; jusText/Trafilatura/Readability handle high-level content filtering. Next, we'll see how to integrate these in a Python solution.

Implementation: Loading and Cleaning HTML (Code Example)

Below is a **complete Python example** demonstrating how to fetch a JavaScript-rendered page using ScrapingFish and then clean it. This example uses **BeautifulSoup** for initial parsing and tag removal, and **jusText** for boilerplate filtering, focusing on English content. (You would replace `YOUR_API_KEY` and `TARGET_URL` with your ScrapingFish API key and the URL of the page to scrape.) Links to documentation are provided in comments for clarity.

python

```
import requests
from bs4 import BeautifulSoup
import justext

# 1. Fetch the dynamic page content via ScrapingFish API
api_key = "YOUR_API_KEY"  # ScrapingFish API key
target_url = "https://example.com/some-business-page"  # The URL of the page to
scrape

params = {
    "api_key": api_key,
    "url": target_url,
    "render_js": True  # Enable JS rendering to load dynamic content
:contentReference[oaicite:9]{index=9}
}
response = requests.get("https://scraping.narf.ai/api/v1/", params=params)
html_content = response.content  # The full HTML (bytes) of the page, including JS-
rendered content

# 2. Parse the HTML content with BeautifulSoup (using lxml parser for robustness)
```

```
soup = BeautifulSoup(html_content, "lxml")

# 3. Remove unwanted elements by tag name (scripts, styles, navbars, footers, etc.)
for tag in soup(["script", "style"]):
    tag.decompose()  # remove script and style entirely
for tag in soup(["header", "footer", "nav", "aside"]):
    tag.decompose()  # remove common layout sections like header, footer, nav, sidebar

# Optionally, remove elements by specific class or id patterns (e.g., ads or banners)
for ad in soup.find_all(attrs={"class": lambda c: c and "advertisement" in c.lower()}):
    ad.decompose()
for ad in soup.find_all(id=lambda i: i and i.lower().startswith("ad_")):
    ad.decompose()
# (The above are examples; you can adjust the filtering criteria based on the page's HTML structure.)

# 4. Use jusText to remove boilerplate and keep main textual content
paragraphs = justext.justext(str(soup), justext.get_stoplist("English"))
clean_chunks = []
for para in paragraphs:
    if not para.is_boilerplate:  # not classified as boilerplate (navigation/menu/etc.)
        text = para.text.strip()
        if text:  # if non-empty
            clean_chunks.append(text)

# At this point, clean_chunks is a list of textual paragraphs in English that jusText considered content.

# 5. Output the cleaned content.
# Option A: Join as plain text paragraphs
clean_text = "\n\n".join(clean_chunks)
print("Cleaned Text Content:\\n", clean_text)

# Option B: Reconstruct minimal HTML with basic structure (e.g., wrap each paragraph in <p> tags)
clean_html = "<div>\n" + "\n".join(f"<p>{para}</p>" for para in clean_chunks) + "\n</div>"
print("Cleaned HTML Content:\\n", clean_html)
```

In the code above, we first retrieve the page. The `requests.get` call to ScrapingFish includes `render_js=True`, which ensures the returned HTML has any JavaScript-generated content. (According to ScrapingFish's documentation, JavaScript rendering must be explicitly enabled .) The response content will be the raw HTML of the page after a headless browser renders it.

Next, we parse the HTML with BeautifulSoup. This gives us a DOM tree (`soup`) to work with. We then systematically remove tags that are not useful for content: all `<script>` and `<style>` tags (which hold scripts and CSS), as well as semantic sections like `<header>`, `<footer>`, `<nav>`, and `<aside>` that typically contain navigation links, headers/footers, or sidebars. We also included an example of removing elements by class or id if they contain substrings like `"advertisement"` or start with `"ad_"` (a simple way to catch ad banners, though in practice you might refine these rules). At this stage, we've dropped a lot of the obvious clutter from the DOM.

Then, we convert the cleaned soup to a string and feed it into **jusText**. JusText will analyze the remaining HTML and classify each textual fragment as either content or boilerplate, using heuristics and an English stopword list. We iterate through the `paragraphs` returned by `justext.justext()` and collect only those where `para.is_boilerplate` is False – i.e., the segments jusText deems as main content. By using the English stoplist, jusText focuses on text that looks like English sentences and filters out fragments that don't match typical prose (which often catches navigation link lists, repetitive menus, etc.) . The result is a list of clean text paragraphs (`clean_chunks`).

Finally, the code shows two ways to output the cleaned content:

- **Option A:** Join the paragraphs with blank lines to produce plain text output. This is useful if you just need the text data (for example, for indexing or analysis).
- **Option B:** Wrap each paragraph in `<p>` tags and enclose in a `<div>` to produce a minimal HTML snippet. This retains some HTML structure, which might be useful if you want to display the content in a browser or keep formatting like line breaks. You could further enhance this by preserving certain tags (for example, if the original had `<h1>` headings or `<table>` with contact info, one could integrate that into the output as well). The idea is that the cleaned HTML contains **only the informative sections** and nothing like scripts or navigation menus.

Note: In some cases, extremely short pieces of content (like a phone number or address line) might be classified as boilerplate by jusText (because they aren't full sentences). If such pieces are crucial (e.g. a contact number), you may need to post-process or adjust the strategy – for example, explicitly allow certain small `<div>` or `<span>` that appear in a "Contact" section, or use the scraping API's extraction features (discussed below). In our example, we focus on paragraph-style content.

Rationale for Library Choices and Structure

ScrapingFish + Requests: Using ScrapingFish simplifies the process of dealing with dynamic content. It **renders JavaScript in a real browser environment** and returns the final HTML, saving us from running a local headless browser like Selenium. The integration is straightforward (just an HTTP GET with the API endpoint), and it handles anti-bot measures as well . This approach lets us retrieve the same HTML a user's browser would see after all scripts run.

BeautifulSoup (with lxml parser): Once we have the HTML, we need to parse and manipulate it easily. BeautifulSoup is "a Python library for pulling data out of HTML...providing idiomatic ways of navigating and modifying the parse tree," which can save us a lot of time . Using BS4 with the lxml parser gives us speed and leniency (it can parse messy HTML from the web). The code structure uses BS4 to remove obvious unwanted nodes. This step is transparent and controllable – we explicitly specify which tags to drop. We can easily tweak this list of tags or add specific classes/ids as needed for different pages.

jusText for boilerplate removal: After initial cleaning, jusText provides a powerful layer of content filtering. It was chosen because *jusText is specifically designed to preserve main textual content while removing boilerplate like navigation links, headers, and footers* . By using jusText (with the appropriate language stoplist), we leverage well-tested heuristics to catch residual clutter that simple tag removal might miss (for example, a list of dozens of city links at the bottom, or a sidebar of related items that wasn't in a `<aside>` tag). JusText helps ensure that what remains is predominantly the core information (full sentences and meaningful text) rather than template text. In our code, jusText nicely complements the manual removal from the previous step.

Alternate extraction libraries: We mentioned Trafilatura and Readability as alternatives. These could have been used to streamline the process – for instance, calling `trafilatura.extract(html_content, output_format="html", target_language="en")` could fetch the main content in one call. Trafilatura is a robust choice because it *focuses on actual content and avoids noise from recurring page elements* ; it even handles things like extracting tables if `include_tables=True` (which is useful if business info is in tables). We opted for a more step-by-step approach in code for clarity, but in a real project you might choose Trafilatura to do the heavy lifting. Similarly, using `readability-lxml`'s `Document` on the HTML and calling `doc.summary()` would return cleaned HTML of the main article/text , which can be convenient for article-like pages. The rationale for our chosen stack is to demonstrate control and transparency in the cleaning process, but the best choice can depend on the exact page structure and the content you need.

Maintaining English focus: Both jusText and Trafilatura support specifying the language (English in our case) for their internal algorithms. This is important because boilerplate removal often relies on detecting common words and patterns of a language – using the English stoplist helps identify sentences vs. navigation text accurately . Our solution passes English to jusText, and we could similarly enforce English in Trafilatura (`target_language='en'`) if using that. This ensures that, for example, if the page had snippets in another language, they might be ignored as not relevant.

Output format considerations: We showed two output forms. Delivering **clean HTML** can be useful if the cleaned data will be presented on a webpage or needs some formatting (preserving `<p>` tags, basic lists or tables for contacts, etc.). On the other hand, **plain text** might be better for downstream NLP processing or storage in a database as raw content. The solution structure is flexible: because we isolate the content in `clean_chunks` (as text), we can easily format it as needed (HTML, JSON, etc.). For example, one could create a JSON with fields like `"name": ...`, `"contact": ...` by further parsing the clean text if the format is known.

ScrapingFish extraction rules (alternative approach): It's worth noting that ScrapingFish itself provides an **extraction feature** where you can supply CSS selectors and get back specific content in JSON format . For instance, if we know the page's company name is always in an `<h1>` with class "biz-name" and the phone number in a `<span>` with id "phone", we could let ScrapingFish return those directly, skipping the full HTML cleaning. However, that method requires prior knowledge of the page structure and is less flexible for arbitrary pages. Our approach, in contrast, is more general-purpose: it tries to programmatically discern the important content without hard-coding selectors, which is preferable when scraping many different pages or sites.

In summary, the combination of **ScrapingFish** for dynamic content retrieval, **BeautifulSoup/lxml** for HTML parsing and targeted removals, and **jusText** (or similar extraction libraries) for intelligent content filtering provides a robust solution. This setup leverages powerful open-source tools at each stage: we get the benefit of a rendered page, we programmatically strip out noise, and we rely on proven algorithms to separate the wheat (business-relevant text) from the chaff (boilerplate). The result is a clean, focused HTML/text output that contains only the information we care about, ready for further use or analysis.

Sources:

- ScrapingFish Documentation – *API usage and JavaScript rendering*
- BeautifulSoup Documentation – *HTML parsing and parse-tree navigation*
- jusText GitHub README – *Boilerplate removal (navigation, headers, footers) and preserving main text*
- Trafilatura GitHub README – *Purpose of extracting text and avoiding noise (headers/footers)*
- readability-lxml PyPI – *Arc90 Readability algorithm for main content extraction*
- ScrapingFish Extraction Rules – *Using CSS selectors to extract content as JSON*

Sources