📘 Scrapy Project Documentation: Matweb, ASM, Matdat Scraper Suite

---

🚀 Part 1: High-Level Overview

◆ Project Objective

This Scrapy-based project is designed to scrape detailed material property data from multiple sources (MatWeb, ASM, Matdat, etc.) using dedicated spiders for each source. The architecture is modular and scalable, allowing future integration of additional material databases with minimal changes.

◆ Core Features

- Input-driven scraping based on JSON configurations
- Reusable spider base class for input processing and utilities
- Structured item modeling for consistency
- Flexible pipelines for output formatting and cleanup
- Supports multiple output formats (primarily CSV)
- Minimal coupling between spiders

◆ Data Flow (Summary)

```
Input JSON --> Spider --> Item --> Pipeline --> CSV Output
           ↘ Utility/Helpers ↗
```

□ Part 2: Directory & Module Overview

Path	Role
`matdata/base_spider.py`	Base class providing shared spider logic
`matdata/matweb_spider.py`	Spider for scraping `matweb.com`
`matdata/asm_spider.py`	Spider for scraping `asm.matweb.com`
`matdata/matdat_spider.py`	Spider for scraping `matdat.com`
`matdata/tpsx_spider.py`	Placeholder for future TPSX scraper
`matdata/pipelines.py`	Cleans, flattens, and exports scraped data
`matdata/items.py`	Defines `MaterialItem` data schema

Path	Role
`matdata/utils.py`	Common functions for text cleaning, parsing, and flattening
`input/*.json`	Source-specific config files with material names/URLs
`scrapy.cfg`	Declares project and module mapping for Scrapy
`settings.py`	Controls concurrency, logging, user agent, export settings

---

□ Part 3: BaseSpider – `base_spider.py`

Purpose:

Provides shared functionality and enforces structure for all spiders.

Key Functionalities:

- **Input loader (`load_input`)**
  Loads JSON from `/input/{spider_name}_input.json` into memory.
- **Start URLs**
  Dynamically initialized from input file based on material names or links.
- **Request Wrapper**
  Uses `start_requests()` to dispatch requests with error resilience.
- **Retry Logic**
  Custom `handle_failure()` method to retry failed requests a configurable number of times.
- **Abstract parse method**
  Each child spider must override `parse_material` to parse material-specific pages.

Critical Points:

- Failing to define `self.input_file` in child spiders breaks input loading.
- Parsing logic is abstracted — child spiders have full control over `response`.

🕷 `matweb_spider.py` — Spider for scraping MatWeb

📌 Objective

Scrapes material property data from matweb.com.
For each material name provided in `input/matweb_input.json`, the spider searches for matching pages and extracts structured data from the result.

□ Class Definition

```
class MatwebSpider(BaseSpider):
    name = "matweb_spider"
    allowed_domains = ["matweb.com"]
```

- Inherits from `BaseSpider` to reuse `load_input()` and `start_requests()`.
- Automatically loads search queries from `matweb_input.json`.

📥 `start_requests()` Inherited

Each input is interpreted as a material name. It generates a search query URL:

```
https://www.matweb.com/search/SearchResults.aspx?searchtext=<material_name>
```

Each URL is yielded as a Scrapy request and handled by `parse()`.

🔍 `parse()` – Handle search results page

```
def parse(self, response):
    result_links =
response.xpath("//table[@class='results']//a/@href").extract()
```

- Extracts all result links from search result page.
- If no links are found, logs a message and moves on.
- For each valid link, generates a new request to that material detail page → handled by `parse_material`.

```
yield scrapy.Request(url=response.urljoin(link),
callback=self.parse_material)
```

📄 `parse_material()` – Main content scraper

This is the most important method — responsible for parsing the material property table.

```
def parse_material(self, response):
    tables = response.xpath("//table[@class='specs_table']")
```

- Finds all material property tables (HTML table rows with name-value pairs).
- Each table is processed row-by-row to extract:
    - `property_name`
    - `value`

- `condition` (if exists)

These are then structured into a dictionary using utility methods.

🔧 Utilities and Cleaning

```
from ..utils import clean_text, flatten_dict
```

- `clean_text`: Strips whitespace, newlines, non-breaking spaces.
- `flatten_dict`: Flattens nested dictionaries into a flat key-value mapping.

Used to prepare clean, machine-friendly keys and values.

□ Yielding Items

Each parsed page yields a single `MaterialItem`:

```
MaterialItem(
    name=material_name,
    source="matweb",
    properties=flattened_props,
    ...
)
```

`properties` is a flat dictionary of `property_name: value`.

⚠ Edge Cases Handled

- Missing or malformed search result tables
- Skips empty or irrelevant rows
- Logs warning for invalid structures
- Relies on `BaseSpider.handle_failure()` for retrying failed material pages

🔁 Flow Summary (DFD-style logic):

```
matweb_input.json → start_requests() → search page (parse)
→ each link → parse_material → extract tables → yield item
→ item goes to pipeline
```

🕷️ `asm_spider.py` — Spider for scraping ASM MatWeb

📌 Objective

Extract material composition and property data from the ASM MatWeb portal.
ASM provides structured pages for specific standards like ASTM/ISO, and this spider is tailored to parse those.

🔧 Class Definition

```
class AsmSpider(BaseSpider):
    name = "asm_spider"
    allowed_domains = ["asm.matweb.com"]
```

- Inherits from `BaseSpider` to load material names from `asm_input.json`.

🔁 `start_requests()` Inherited

For each material name in input JSON:

```
https://asm.matweb.com/search/search.aspx?searchtext=<material_name>
```

- Sends a request to the search page.
- Callback is `parse()`.

🔍 `parse()` – Handling Search Results

```
def parse(self, response):
    result_links = response.xpath("//a[contains(@href,
'datasheet.aspx')]/@href").extract()
```

- Parses search results and extracts links to material detail pages.
- If no results found: logs a warning and returns.
- Each result link is followed via `parse_material`.

```
yield scrapy.Request(url=response.urljoin(link),
callback=self.parse_material)
```

📄 `parse_material()` – Extracting Properties

General Flow:

- Extracts the **material name, description, and property tables**.
- Tables are structured, with headings indicating property categories.

Example XPath logic:

```
tables = response.xpath("//table[@class='specs_table']")
for table in tables:
    category = table.xpath(".//th/text()").get()
    rows = table.xpath(".//tr")
```

- `category`: Groups related properties (e.g., "Thermal", "Mechanical").
- Each row typically has:
  - First `td`: Property name
  - Second `td`: Value (may include condition)

It uses robust selectors to ensure even partially malformed rows are handled.

🧹 Cleaning & Structuring

Similar to `matweb_spider.py`, it uses:

- `clean_text()` → Clean strings
- `flatten_dict()` → Prepare flattened key-value output

Output is cleaned and normalized into a flat dictionary.

🎁 Item Yielding

Each result yields a `MaterialItem`:

```
MaterialItem(
    name=name,
    source="asm",
    description=description,
    properties=flattened_properties,
    ...
)
```

- `description` is optionally extracted from the header area.
- `properties` includes all relevant property values.

□ Edge Cases Handled

- Empty search results
- Tables with merged headers or missing columns
- Pages without identifiable properties

🔁 Flow Summary

```
asm_input.json → start_requests()
→ search.aspx?searchtext=<name> → parse() → result page(s)
→ parse_material() → extract + clean → yield MaterialItem → pipeline
```