# Scrapple Documentation

***Release***

**Alex Mathew, Harish Balakrishnan**

**Apr 25, 2018**

# Contents

The Internet is a huge source of information. Several people may use data from the Internet to perform various activities, like research or analysis. Data extraction is a primary step involved in data mining and analysis. Extracting content from structured web pages is a vital task to be performed when the Internet is the principal source of data.

The current standards in web structure involve the use of CSS selectors or XPath expressions to select particular tags from which information can be extracted. Web pages are structured as element trees which can be parsed to traverse through the tags. This tree structure, which represents tags as parent/children/siblings, is very useful when tags should be represented in terms of the rest of the web page structure.

Scrapple is a project aimed at designing a framework for building web content extractors. Scrapple uses key-value based configuration files to define parameters to be considered in generating the extractor. It considers the base page URL, selectors for the data to be extracted, and the selector for the links to be crawled through. At its core, Scrapple abstracts the implementation of the extractor, focussing more on representing the selectors for the required tags. Scrapple can be used to generate single page content extractors or link crawlers.

# CHAPTER 1

# Overview

Web content extraction is a common task in the process of collecting data for data analysis. There are several existing frameworks that aid in this task. In this chapter, a brief introduction of Scrapple is provided, with instructions on setting up the development machine to run Scrapple.

## 1.1 Introducing Scrapple

The Internet is a huge source of information. Several people may use data from the Internet to perform various activities, like research or analysis. However, there are two primary issues involved with using data from the Internet :

- You may not have any way to get information from a particular website, i.e, it may not provide an API for accessing the data.
- Even if an API is provided, it may not give all the data needed. It is possible that there may be some data that is present on the web interface, but not provided through the API.

This is where web scrapers and web crawlers come in.

### 1.1.1 Web scrapers & web crawlers

**Web scrapers**, also called **extractors**, are used to extract content from any particular page. They may use CSS Selectors or XPath expressions to point to a particular tag on the HTML structure of the page, and extract the content from that tag. The content extracted could be text from `<div>` tags, links from `<a>` tags, and so on.

**Web crawlers** are scripts that go through multiple links from a single base page. Given a base URL, it uses this page as an index page to many different pages. It goes through each of these pages, extracting the required content along the way.

Scrapers and crawlers can be written to extract necessary content from any page that you would need information from.

### 1.1.2 How Scrapple helps

Scrapple helps to reduce the hassle in manually writing the scripts needed to extract the required content. It involves the use of a configuration file that specifies property-value pairs of various parameters involved in constructing the required script.

The configuration file is a JSON document, consisting of the required key-value pairs. The user specifies the base URL of the page to work on, and also the tags of the data to be extracted. The user has a choice between CSS selectors and XPath expressions for specifying the target tags. Once the target tags have been specified, the other parameters are filled and the configuration file is completed. This configuration file is used by Scrapple to generate the required script and the execution is performed to generate the output of the scraper as a CSV/JSON document (depending on the argument passed while running the script). Thus, the user can obtain data they need without having extensive programming expertise to manually write the scripts required.

### 1.1.3 The inspiration behind Scrapple

Scrapple is based on the best ideas involved in two projects :

- Scrapy : Scrapy is an application framework, designed to build web spiders that extract structured web data.
- Ducky : Ducky is a semi-automatic web wrapper, which uses a configuration file to define extraction rules, and extract data accordingly.

## 1.2 Review of existing systems

Data extraction process from the web can be classified based on the selectors used. Selectors can be CSS or XPath expressions. CSS selectors are said to be faster and are used by many browsers. Ducky[1] uses CSS selectors for extracting data from pages that are similarly structured.

On the other hand, XPath expressions are more reliable, handles text recognition better and a powerful option to locate elements when compared to CSS selectors. Many researches are going

[1] Kei Kanaoka, Yotaro Fujii and Motomichi Toyama. Ducky: A Data Extraction System for Various Structured Web Documents. In Proceedings of the 18th International Database Engineering & Applications Symposium, IDEAS '14, pages 342-347, New York, NY, USA, 2014. ACM

on presently in this topic. Oxpath[2] provides an extension for XPath expressions. The system created by V. Crescenzi, P. Merialdo, and D. Qiu[3] uses XPath expressions for locating the training data to create queries posed to the workers of a crowd sourcing platform.

Systems like Ducky and Deixto[4] use the concept of Configuration files where the user inputs the simple details like base pages, a "next" column if there are multiple pages to be parsed. Deixto uses the concept of tag filtering where the unnecessary html tags can be ignored when the DOM (Document Object Model) tree is created.

Scrapy[5], an open source project, provides the framework for web crawlers and extractors. This framework provides support for spider programs that are manually written to extract data from the web. It uses XPath expression to locate the content. The output formats of Ducky and Scrapy include XML, CSV and JSON files.

## 1.3 System requirements

Scrapple is a Python package/command line tool which runs on all operating systems that support Python. This includes :

- Windows XP
- Windows Vista
- Windows 7
- Windows 8.x
- Common Linux distros : Ubuntu/Xubuntu/Lubuntu, Fedora, Mint, Gentoo, openSUSE, Arch Linux etc.
- OS X

The basic requirements for running Scrapple are:

- Python 2.7 or 3.x
- *pip* or *easy_install* for installing the necessary Python packages [*pip* is the recommended choice]

Scrapple depends on a number of Python packages for various parts of its execution :

[2] T.Furche, G.Gottlob, G.Grasso, C.Schallhart, and A.Sellers. Oxpath: A language for scalable data extraction, automation, and crawling on the deep web. The VLDB Journal, 22(1):47–72, Feb. 2013

[3] V.Crescenzi, P.Merialdo, and D.Qiu. Alfred: Crowd assisted data extraction. In Proceedings of the 22nd International Conference on World Wide Web Companion, WWW '13 Companion, pages 297–300, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[4] F.Kokkoras, K.Ntonas, and N.Bassiliades. Deixto: A web data extraction suite. In Proceedings of the 6th Balkan Conference in Informatics, BCI '13, pages 9–12, New York, NY, USA, 2013. ACM.

[5] Scrapy: A fast and powerful scraping and web crawling framework.

- requests : The HTTP library. The requests library is used to make HTTP requests to load the required web pages.
- lxml : The web scraping library. The lxml library is used to parse the *element tree* and extract the required content.
- cssselect : The CSS selector library. cssselect works in tandem with lxml to handle CSS Selector expressions.
- docopt : The command line parser. The docopt library is used to parse the command line interface input based on the CLI usage specification in the docstring.
- Jinja2 : The templating engine. Jinja2 is used to create skeleton *configuration file* and generated Python scraper scripts.
- Flask : The web micro-framework. The web interface to edit configuration files runs on Flask.
- colorama : The output formatter. colorama is used to format the various sections of the command line output.

## 1.4 Install Scrapple

The *requirements* covers the system requirements in detail.

If you're running Ubuntu, install the necessary C libraries for the lxml module.

```
$ sudo apt-get install libxml2-dev libxslt-dev python-dev
lib32z1-dev
```

If you're running any other Linux distro, follow the standard install procedures and install these libraries.

You may not have to do this on your Windows machine.

Install the requirements for running Scrapple with

```
$ pip install -r requirements.txt
```

If this fails because of the access privileges, run the command with `sudo`.

You can then install Scrapple with

`$ pip install scrapple` or `$ sudo pip install scrapple`

To verify that Scrapple has been installed correctly, try the `scrapple` command from the command line.

```
$ scrapple --version
```

This should display the version of Scrapple installed.

*Introducing Scrapple* An introduction to Scrapple

*Review of existing systems* A review of existing systems

*System requirements* Hardware and software requirements to run Scrapple

*Install Scrapple* Instructions for installing Scrapple and the required dependencies

CHAPTER 2

# Concepts

Creating web content extractors requires a good understanding of the following topics :

- *Web page structure*
- *Selector expressions*
- *Data formats*

In this chapter, a brief overview of the concepts behind Scrapple is given.

## 2.1 Web page structure

### 2.1.1 Structure of a Web page

The main elements that comprise a web page are :

- DOCTYPE: This lets the browser know the type of markup language the page is written in.
- Document Tree: We can consider a page as a document tree that contain any number of branches.
- HTML: This is the root element of the document tree and everything that follows is a child node. HTML has two descendants – HEAD and BODY
- HEAD: It contains the title and the information of the page.
- BODY: It contains the data displayed by the page.

### 2.1.2 ElementTree

The Element type [6] is a data object that can contain tree-like data structures.

The ElementTree wrapper [6] type adds code to load web pages as trees of Element objects. An element consists of properties like a tag(identify the element type), number of attributes, text string holding the textual content and the number of child nodes.

To create a tree, we create the root element and add children elements to the root element. A method called Subelement can be used for creating and adding an element to the parent element. Few methods that are provided to search for Subelements are as follows:

- find(pattern) – Return the first subelement matching the pattern
- findtext(pattern) – Returns the value of the text attribute of the first subelement matching the pattern
- findall(pattern) – Return a list matching the pattern
- getiterator(tag) - Return a list matching the tag attribute
- getiterator() – Return a list of all the Subelements

## 2.2 Selector expressions

Selector expressions are used to point to specific groups of tags on a web page. They involve using the tag type and tag attributes to select certain sections of the page.

There are two main types of selector expressions that are commonly used, and supported by Scrapple :

- CSS selectors
- XPath expressions

### 2.2.1 CSS Selectors

CSS selectors [7] offer flexible options to design efficient and dynamic tag queries, providing variety of expression and simplicity. They use tag names and tag attributes to represent the traversal through the DOM tree.

For example,

`div#sidebar dl dd a` refers to an anchor tag that is under a definition list in the `<div>` with a id ‘sidebar’.

In recent times, the use of CSS selectors is becoming more common because of the development of Javascript libraries (like jQuery and React.js) which use CSS selectors to access particular tags in the web page structure.

CSS selectors are faster in many browsers. They rely on the structure as well as the attributes of the page to find elements. They provide a good balance between structure and attributes. Though the property values cannot be specified as expressions, it is widely used due to its simplicity and flexibility. They can traverse down a DOM(Document Object Model) according to the path specified in the selector expression.

A detailed study on CSS selectors can be found on W3C.

### 2.2.2 XPath Selectors

XPath [8] is used for navigating documents and selecting nodes. It is based on a tree representation [9] of the document it is to be traversed. Since it can traverse both up and down the DOM, it is used widely used for easy navigation through the page to locate the elements searched for. Here are few simple examples of XPath expressions and their meaning.

- `/html/body/p[1]` : This finds the first p inside the body section of the html
- `//link` : This selects all the link tags in the page
- `//div[@id="commands"]` : This selects the div elements which contain the id=”commands”

XPath expressions can be simplified by looking up for path having unique attributes in the page. For instance, consider the expression

```
//*[@id="guide-container"]/div/ul/li[1]/ \
div/ul/li[@id="what_to_watch-guide-item"]/a
```

This can be simplified further as `//*[@id="what_to_watch-guide-item"]/a` based on the unique id=”what_to_watch-guide-item” attribute used.

They can also be simplified by shortening the reference path. For instance, consider the same expression as the previous example. This expression can be shortened as `//*[@id="guide-container"]//li[@id="what_to_watch-guide-item"]/a` where the ‘//’ refers to the shortened path.

There are several other advanced methods of representing XPath expressions, like expressing the ancestors or siblings, normalizing spaces etc.

A detailed study on XPath expressions can be found on W3C.

## 2.3 Data formats

There are several data formats that are used to handle data. This includes XML, CSV, JSON, etc. Scrapple provides support for storing extracted data in two formats :

- Javascript Object Notation (JSON)
- Comma Separated Values (CSV)

### 2.3.1 JSON

Javascript Object Notation (JSON) files are easy to understand and create. They are easy to parse through, understand and write. It is a language independent format and hence many of the APIs use them as a data-interchange format.

Few data types in JSON are :

- Object: It is an unordered set of name/value pairs.
- Array: It is a set of values of same data type. It is enclosed in a square bracket and the name-value pairs are separated by a comma.
- Name: It is the field that describes the data.
- Value: It is the input data for the name attribute. It can be a number, a Boolean value(true or false), a character(inserted between single quotes) or a string(inserted between double quotes).

For example,

```
{

        "subject": "Computer Science",
        "data": [
        # Array

                {
                # Object

                        "name": "John",          # String
                        "marks": 96,             # Integer
                        "passed": true           # Boolean

                },

                {

                        "name": "Doe",
```

```
                        "marks": 33,
                        "passed": false

                }

        ]

}
```

### 2.3.2 CSV

Comma Separated Values (CSV) files consists of tabular data where the fields are separated by a comma and the records by a line. It is stored in plain-text format. CSV files are easy to handle and manipulate.

For example,

| Name | Marks | Grade | Promotion |
|---|---|---|---|
| John | 96 | O | True |
| Doe | 45 | F | False |

can be represented as,

```
John,96,O,True
Doe,45,F,False
```

***Web page structure*** The basics of web page structure and element trees

***Selector expressions*** An introduction to tag selector expressions

***Data formats*** The primary data formats involved in handling data

# CHAPTER 3

# The Scrapple framework

This section deals with how Scrapple works - the architecture of the Scrapple framework, the commands and options provided by the framework and the specification of the configuration file.

## 3.1 Scrapple architecture

Scrapple provides a command line interface (CLI) to access a set of commands which can be used for implementing various types of web content extractors. The basic architecture of Scrapple explains how the various components are related.

- ***Command line input*** The command line input is the basis of definition of the implementation of the extractor. It specifies the project configuration and the options related to implementing the extractor.
- ***Configuration file*** The configuration file specifies the rules of the required extractor. It contains the selector expressions for the data to be extracted and the specification of the link crawler.
- ***Extractor framework*** The extractor framework handles the implementation of the parsing & extraction. The extractor framework follows the following steps :
  - It makes HTTP requests to fetch the web page to be parsed.
  - It parses through the *element tree*.
  - It extracts the required content, depending on the extractor rules in the configuration file.

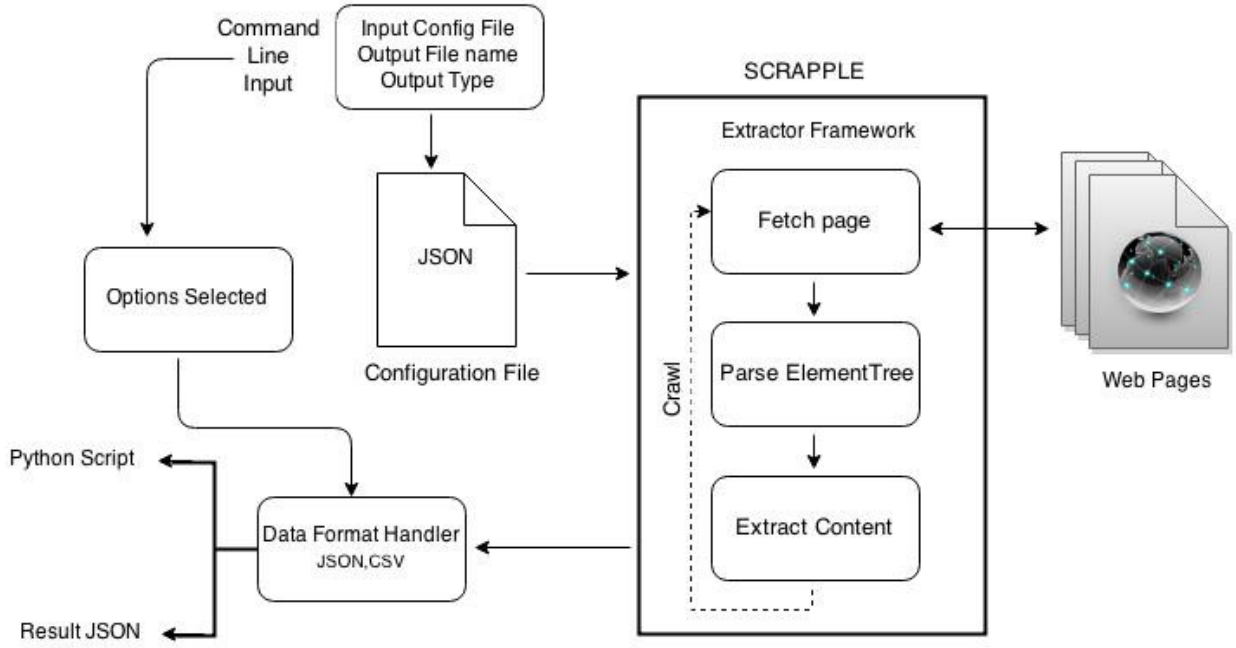


Fig. 3.1: Scrapple architecture

  - In case of crawlers, this process is repeated for all the pages that the extractor crawls through.

- ***Data format handler*** According to the options specified in the CLI input, the extracted content is stored as a CSV document or a JSON document.

## 3.2 Scrapple commands

The four commands provided by the Scrapple CLI are

- genconfig
- generate
- run
- web

### 3.2.1 genconfig

The `genconfig` command is used to create a skeleton *configuration file*, which can be used as the base for further writing the necessary configuration file. This makes it easier to understand the structure of the key/value-based configuration file, and provide the necessary options.

The two positional arguments for the genconfig command are :

- The project name
- The base URL

The `genconfig` command creates a basic configuration file with the provided base URL, and creates it as "<project_name>.json".

The two optional arguments for the genconfig command are :

- The type of extractor
- The selector type
- Number of crawl levels

The extractor could be a scraper or a crawler, and this can be specified in the –type argument. By default, it is a scraper. When the crawler option is provided, it adds the "next" parameter in the skeleton configuration file.

With the –selector option, the selector type to be used can be specified. This can be "css" or "xpath". By default, it is "xpath".

For crawler configuration specifications, the number of levels of the crawler can be specified using the –levels argument. By default, it is 1. According to the number of levels specified, *nested "next" fields* are created.

Examples :

- `$ scrapple genconfig pyvideo http://pyvideo.org/category` creates pyvideo.json, which contains the skeleton configuration file for a scraper which uses XPath expressions as the selector.
- `$ scrapple genconfig pyvideo http://pyvideo.org/category --type=crawler` creates pyvideo.json, which contains the skeleton configuration file for a single level crawler which uses XPath expressions as the selector.
- `$ scrapple genconfig pyvideo http://pyvideo.org/category --type=crawler --selector=css --levels=2` creates pyvideo.json, which contains the skeleton configuration file for a 2 level crawler which uses CSS selector expressions as the selector.

### 3.2.2 generate

The `generate` command is used to generate the Python script corresponding to the specifications in the configuration file. This command is used to create the script that replicates the operation of the run command.

The two positional arguments for the generate command are :

- The project name
- The output file name

The project name is the name of the configuration file to be used, i.e, "<project_name>.json" is the configuration file used as the specification. The command creates "<output_file_name>.py" as the generated Python script.

The one available optional argument is :

- The output type

This specifies the output format in which the extracted content is to be stored. This could be "csv" or "json". By default, it is "json".

Examples :

- `$ scrapple generate pyvideo talk1` generates talk1.py based on pyvideo.json, where the extracted data is stored in a JSON document.
- `$ scrapple generate pyvideo talk1 --output_type=csv` generates talk1.py based on pyvideo.json, where the extracted data is stored in a CSV document.

### 3.2.3 run

The `run` command is used to run the extractor corresponding to the specifications in the configuration file. This command runs the extractors and stores the extracted content for later use.

The two positional arguments for the generate command are :

- The project name
- The output file name

The project name is the name of the configuration file to be used, i.e, "<project_name>.json" is the configuration file used as the specification. The command creates "<output_file_name>.json" or "<output_file_name>.csv" which contains the extracted content.

The one available optional argument is :

- The output type

This specifies the output format in which the extracted content is to be stored. This could be "csv" or "json". By default, it is "json".

Examples :

- `$ scrapple run pyvideo talk1` runs the extractor based on pyvideo.json, and stores the extracted content in talk1.json.
- `$ scrapple run pyvideo talk1 --output_type=csv` runs the extractor based on pyvideo.json, and stores the extracted content in talk1.csv.

### 3.2.4 web

The `web` command is an added feature, to make it easier to edit the configuration file. It provides a web interface, which contains a form where the configuration file can be filled. It currently supports only editing configuration files for scrapers. Future work includes support for editing configuration files for link crawlers.

The web interface can be opened with the command

```
$ scrapple web
```

This starts a Flask web app, which opens on port 5000 on the localhost.

## 3.3 Configuration file

The configuration file is the basic specification of the extractor required. It contains the URL for the web page to be loaded, the selector expressions for the data to be extracted and in the case of crawlers, the selector expression for the links to be crawled through.

The keys used in the configuration file are :

- **project_name** : Specifies the name of the project with which the configuration file is associated.
- **selector_type** : Specifies the type of selector expressions used. This could be "xpath" or "css".
- **scraping** [Specifies parameters for the extractor to be created.]
  - **url** : Specifies the URL of the base web page to be loaded.
  - **data** [Specifies a list of selectors for the data to be extracted.]
    - **selector** : Specifies the selector expression.
    - **attr** : Specifies the attribute to be extracted from the result of the selector expression.
    - **field** : Specifies the field name under which this data is to stored.
    - **default** : Specifies the default value to be used if the selector expression fails.
  - **table** [Specifies a description for scraping tabular data.]
    - **table_type** : Specifies the type of table ("rows" or "columns"). This determines the type of table to be extracted. A row extraction is when there is a single row to be extracted and mapped to a set of headers. A column extraction is when a set of rows have to be extracted, giving a list of header-value mappings.
    - **header** : Specifies the headers to be used for the table. This can be a list of headers, or a selector that gives the list of headers.
    - **prefix** : Specifies a prefix to be added to each header.
    - **suffix** : Specifies a suffix to be added to each header.
    - **selector** : Specifies the selector for the data. For row extraction, this is a selector that gives the row to be extracted. For column extraction, this is a list of selectors for each column.
    - **attr** : Specifies the attribute to be extracted from the selected tag.
    - **default** : Specifies the default value to be used if the selector does not return any data.
  - **next** [Specifies the crawler implementation.]
    - **follow_link** : Specifies the selector expression for the `<a>` tags to be crawled through.

The main objective of the configuration file is to specify extraction rules in terms of selector expressions and the attribute to be extracted. There are certain set forms of selector/attribute value pairs that perform various types of content extraction.

Selector expressions :

- CSS selector or XPath expressions that specify the tag to be selected.
- "url" to take the URL of the current page on which extraction is being performed.

Attribute selectors :

- "text" to extract the textual content from that tag.
- "href", "src" etc., to extract any of the other attributes of the selected tag.

***Scrapple architecture*** The architecture of the Scrapple framework

***Scrapple commands*** Commands provided by the Scrapple CLI

***Configuration file*** The configuration file which is used by Scrapple to implement the required extractor/crawler

CHAPTER 4

# Implementation

This section deals with the implementation of the Scrapple framework. This includes an explanation of the classes involved in the framework, the interaction scenarios for each of the commands supported by Scrapple, and utility functions that form a part of the implementation of the extractor.

## 4.1 Scrapple implementation classes

There are two main categories of classes involved in the implementation of extractors on Scrapple :

- *Command classes*
- *Selector classes*

**Command classes** define the execution of the commands provided by Scrapple.

**Selector classes** define the implementation of the extraction through the selector expressions.

The following class diagram can be used to represent the relationship between the various classes.

## 4.2 Interaction scenarios

The primary use cases in Scrapple are the execution of the *commands* provided by the framework. A general idea of the execution of these commands and the relation between the various modules

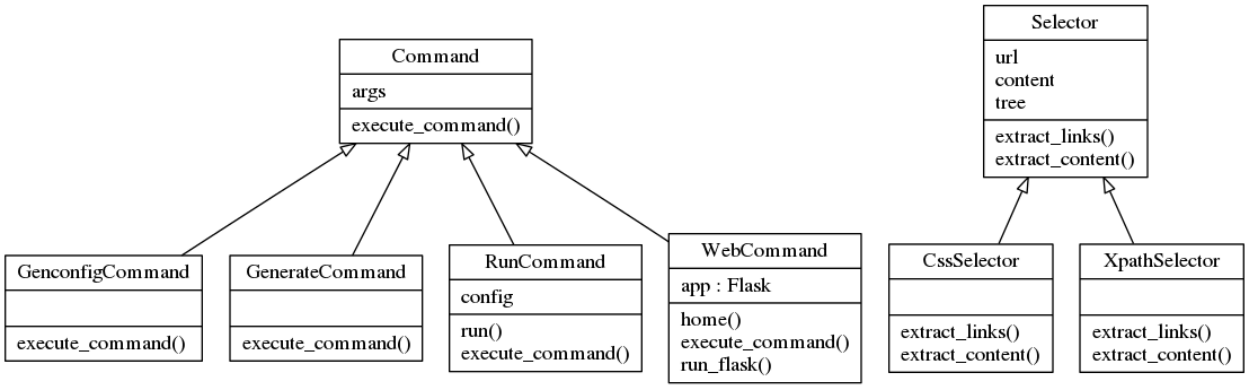


Fig. 4.1: Scrapple class diagram

of the framework can be understood through a study of the interaction scenarios for each of the commands.

Basic sequence diagrams for the execution for each command can be represented as such. A more detailed explanation of the execution of the commands is provided in the *commands implementation* section.

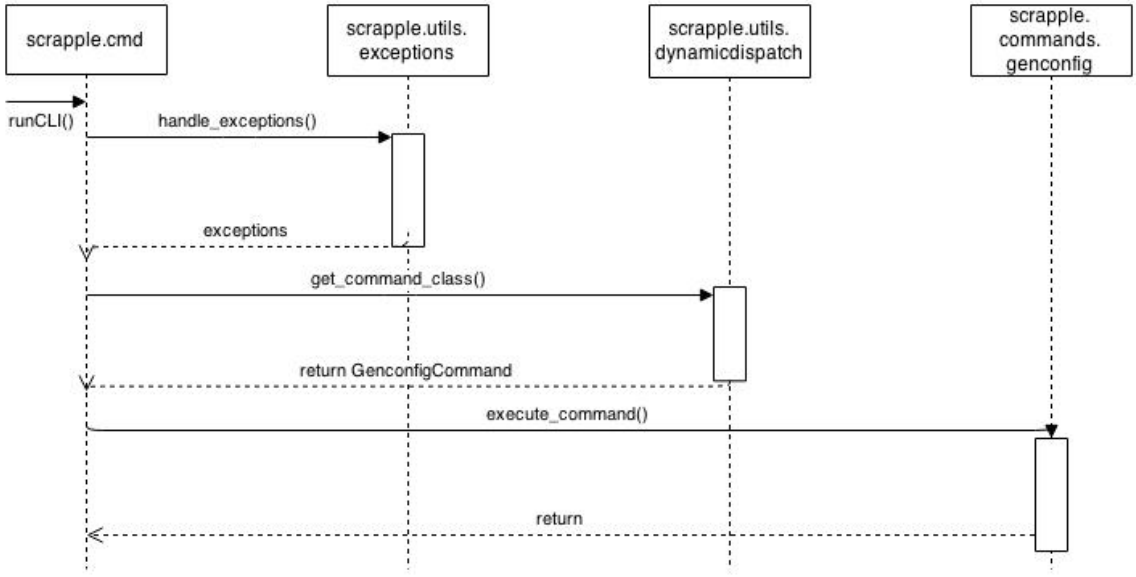


Fig. 4.2: *Genconfig command*

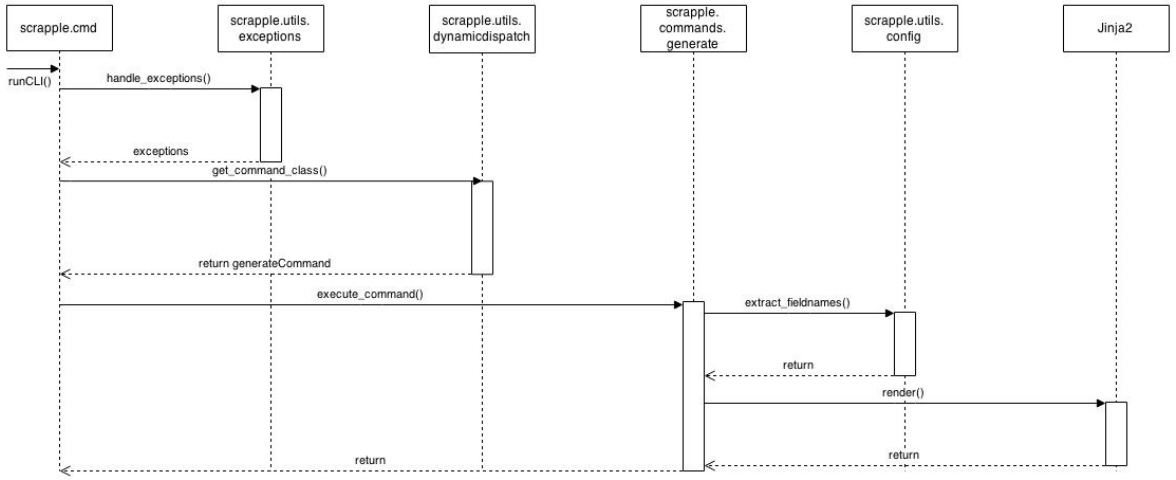


Fig. 4.3: *Generate command*

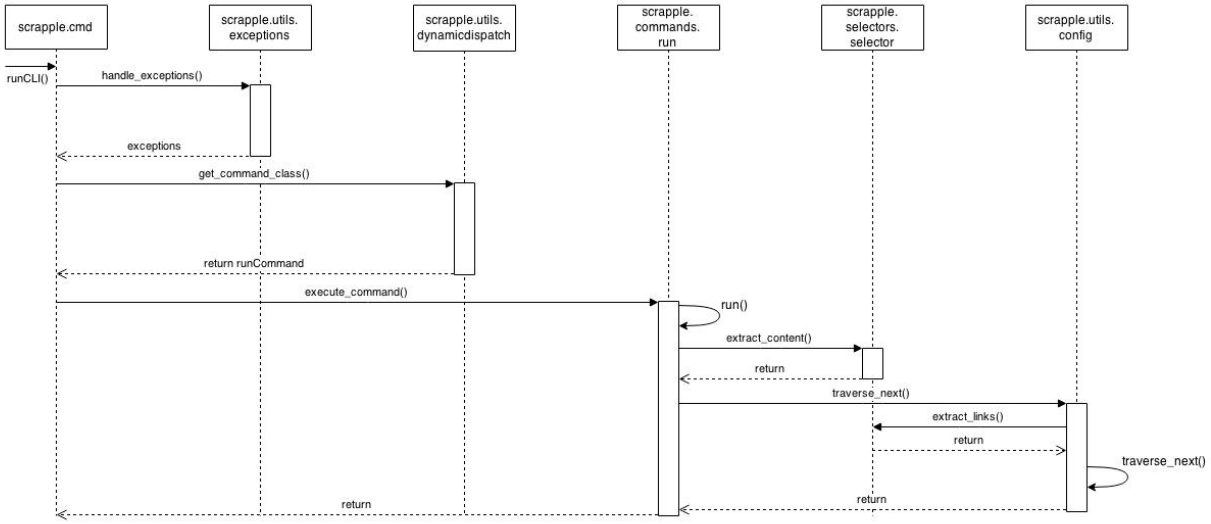


Fig. 4.4: *Run command*

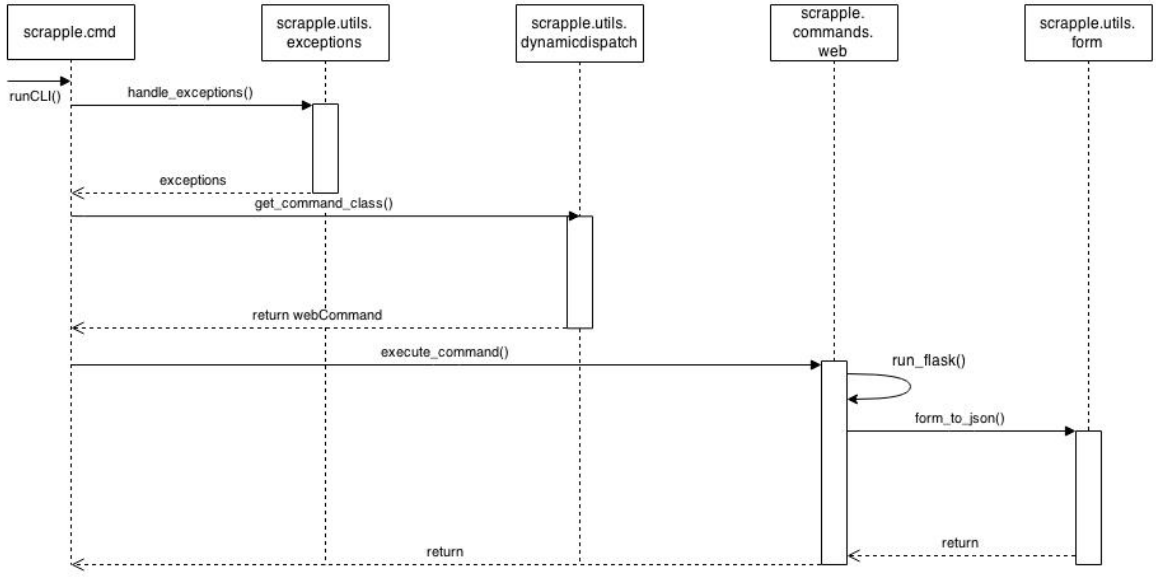


Fig. 4.5: *Web command*

## 4.3 Command line interface

Scrapple is primarily run through a command line interface. The CLI is used to execute the *commands supported by Scrapple*. For a description of the usage of the Scrapple CLI, the help option can be used.

```
$ scrapple --help
```

This presents the usage description and an explanation of the optional arguments provided by the commands.

Usage:

scrapple (-h | –help | –version)

scrapple genconfig <projectname> <url> [–type=<type>] [–selector=<selector>]

scrapple run <projectname> <output_filename> [–output_type=<output_type>]

scrapple generate <projectname> <output_filename> [–output_type=<output_type>]

scrapple web

**Options:**

| | |
|---|---|
| **-h, --help** | Show this help message and exit |
| **--version** | Display the version of Scrapple |

**--type=<type>, -t <type>** Specifies if the script generated is a page scraper or a crawler [default: scraper]

**--selector=<selector>, -s <selector>** Specifies if XPath expressions or CSS selectors are used [default: xpath]

**--output_type=<output_type>, -o <output_type>** Specifies if the generated output is stored as CSV or JSON [default: json]

The `scrapple` tool on the command line is included in the system path when Scrapple is *installed*. When the tool is run, the input on the CLI is parsed by the runCLI() function.

`scrapple.cmd.`**`runCLI`**`()`
The starting point for the execution of the Scrapple command line tool.

runCLI uses the docstring as the usage description for the scrapple command. The class for the required command is selected by a dynamic dispatch, and the command is executed through the execute_command() method of the command class.

This functionality is implemented by the following code block -

```
handle_exceptions(args)
command_list = ['genconfig', 'run', 'generate', 'web']
select = itemgetter('genconfig', 'run', 'generate', 'web')
selectedCommand = command_list[select(args).index(True)]
cmdClass = get_command_class(selectedCommand)
obj = cmdClass(args)
obj.execute_command()
```

## 4.4 Command classes

### 4.4.1 scrapple.commands.genconfig

**`class`** `scrapple.commands.genconfig.`**`GenconfigCommand`**`(`*args*`)`
Defines the execution of *genconfig*

**`execute_command`**`()`
The genconfig command depends on predefined Jinja2 templates for the skeleton configuration files. Taking the –type argument from the CLI input, the corresponding template file is used.

Settings for the configuration file, like project name, selector type and URL are taken from the CLI input and using these as parameters, the template is rendered. This rendered JSON document is saved as <project_name>.json.

### 4.4.2 scrapple.commands.generate

**class** scrapple.commands.generate.**GenerateCommand**(*args*)

Defines the execution of *generate*

**execute_command**()

The generate command uses Jinja2 templates to create Python scripts, according to the specification in the configuration file. The predefined templates use the extract_content() method of the *selector classes* to implement linear extractors and use recursive for loops to implement multiple levels of link crawlers. This implementation is effectively a representation of the traverse_next() *utility function*, using the loop depth to differentiate between levels of the crawler execution.

According to the –output_type argument in the CLI input, the results are written into a JSON document or a CSV document.

The Python script is written into <output_filename>.py - running this file is the equivalent of using the Scrapple *run command*.

### 4.4.3 scrapple.commands.run

**class** scrapple.commands.run.**RunCommand**(*args*)

Defines the execution of *run*

**execute_command**()

The run command implements the web content extractor corresponding to the given configuration file.

The execute_command() validates the input project name and opens the JSON configuration file. The run() method handles the execution of the extractor run.

The extractor implementation follows these primary steps :

1. Selects the appropriate *selector class* through a dynamic dispatch, with the selector_type argument from the CLI input.
2. Iterate through the data section in level-0 of the configuration file. On each data item, call the extract_content() method from the selector class to extract the content according to the specified extractor rule.
3. If there are multiple levels of the extractor, i.e, if there is a 'next' attribute in the configuration file, call the traverse_next() *utility function* and parse through successive levels of the configuration file.
4. According to the –output_type argument, the result data is saved in a JSON document or a CSV document.

### 4.4.4 scrapple.commands.web

**class** scrapple.commands.web.**WebCommand**(*args*)

Defines the execution of *web*

**execute_command**()

The web command runs the Scrapple web interface through a simple Flask app.

When the execute_command() method is called from the *runCLI()* function, it starts of two simultaneous processes :

- Calls the run_flask() method to start the Flask app on port 5000 of localhost
- Opens the web interface on a web browser

The '/' view of the Flask app, opens up the Scrapple web interface. This provides a basic form, to fill in the required configuration file. On submitting the form, it makes a POST request, passing in the form in the request header. This form is passed to the form_to_json() *utility function*, where the form is converted into the resultant JSON configuration file.

Currently, closing the web command execution requires making a keyboard interrupt on the command line after the web interface has been closed.

## 4.5 Selector classes

*Selectors* are used to specifically point to certain tags on a web page, from which content has to be extracted. In Scrapple, selectors are implemented through selector classes, which define methods to extract necessary content through specified selector expressions and to extract links from anchor tags to be crawled through.

There are two selector types that are supported in Scrapple :

- XPath expressions
- CSS selector expressions

These selector types are implemented through the `XpathSelector` and `CssSelector` classes, respectively. These two classes use the `Selector` class as their super class.

In the super class, the URL of the web page to be loaded is validated - ensuring the schema has been specified, and that the URL is valid. A HTTP GET request is made to load the web page, and the HTML content of this fetched web page is used to generate the *element tree*. This is the element tree that will be parsed to extract the necessary content.

### 4.5.1 scrapple.selectors.selector

**class** scrapple.selectors.selector.**Selector**(*url*)

This class defines the basic `Selector` object.

**extract_columns**(*result={}*, *selector=''*, *table_headers=[]*, *attr=''*, *connector=''*, *default=''*, *verbosity=0*, **args*, ***kwargs*)

Column data extraction for extract_tabular

**extract_content**(*selector=''*, *attr=''*, *default=''*, *connector=''*, **args*, ***kwargs*)

Method for performing the content extraction for the particular selector type. If the selector is "url", the URL of the current web page is returned. Otherwise, the selector expression is used to extract content. The particular attribute to be extracted ("text", "href", etc.) is specified in the method arguments, and this is used to extract the required content. If the content extracted is a link (from an attr value of "href" or "src"), the URL is parsed to convert the relative path into an absolute path.

If the selector does not fetch any content, the default value is returned. If no default value is specified, an exception is raised.

**Parameters**

- **selector** – The XPath expression
- **attr** – The attribute to be extracted from the selected tag
- **default** – The default value to be used if the selector does not return any data
- **connector** – String connector for list of data returned for a particular selector

**Returns** The extracted content

**extract_links**(*selector=''*, **args*, ***kwargs*)

Method for performing the link extraction for the crawler. The selector passed as the argument is a selector to point to the anchor tags that the crawler should pass through. A list of links is obtained, and the links are iterated through. The relative paths are converted into absolute paths and a `XpathSelector`/`CssSelector` object (as is the case) is created with the URL of the next page as the argument and this created object is yielded.

The extract_links method basically generates `XpathSelector`/`CssSelector` objects for all of the links to be crawled through.

**Parameters** **selector** – The selector for the anchor tags to be crawled through

**Returns** A `XpathSelector`/`CssSelector` object for every page to be crawled through

**extract_rows**(*result={}*, *selector=''*, *table_headers=[]*, *attr=''*, *connector=''*, *default=''*, *verbosity=0*, **args*, ***kwargs*)
Row data extraction for extract_tabular

**extract_tabular**(*header=''*, *prefix=''*, *suffix=''*, *table_type=''*, **args*, ***kwargs*)
Method for performing the tabular data extraction. :param result: A dictionary containing the extracted data so far :param table_type: Can be "rows" or "columns". This determines the type of table to be extracted. A row extraction is when there is a single row to be extracted and mapped to a set of headers. A column extraction is when a set of rows have to be extracted, giving a list of header-value mappings. :param header: The headers to be used for the table. This can be a list of headers, or a selector that gives the list of headers :param prefix: A prefix to be added to each header :param suffix: A suffix to be added to each header :param selector: For row extraction, this is a selector that gives the row to be extracted. For column extraction, this is a list of selectors for each column. :param attr: The attribute to be extracted from the selected tag :param default: The default value to be used if the selector does not return any data :param verbosity: The verbosity set as the argument for scrapple run :return: A 2-tuple containing the list of all the column headers extracted and the list of dictionaries which contain (header, content) pairs

### 4.5.2 scrapple.selectors.xpath

**class** scrapple.selectors.xpath.**XpathSelector**(*url*)
The `XpathSelector` object defines XPath expressions.

### 4.5.3 scrapple.selectors.css

**class** scrapple.selectors.css.**CssSelector**(*url*)
The `CssSelector` object defines CSS selector expressions.

## 4.6 Utility functions

### 4.6.1 scrapple.utils.dynamicdispatch

Functions related to dynamic dispatch of objects

scrapple.utils.dynamicdispatch.**get_command_class**(*command*)
Called from runCLI() to select the command class for the selected command.

**Parameters** **command** – The command to be implemented

**Returns** The command class corresponding to the selected command

This function is implemented through this simple code block :

```
from scrapple.commands import genconfig, generate, run, web
cmdClass = getattr(eval(command), command.title() + 'Command')
return cmdClass
```

### 4.6.2 scrapple.utils.exceptions

Functions related to handling exceptions in the input arguments

The function uses regular expressions to validate the CLI input.

```
projectname_re = re.compile(r'[^a-zA-Z0-9_]')
if args['genconfig']:
        if args['--type'] not in ['scraper', 'crawler']:
                raise Exception("--type has to be 'scraper' or 'crawler
 ↪'")
        if args['--selector'] not in ['xpath', 'css']:
                raise Exception("--selector has to be 'xpath' or 'css'
 ↪")
if args['generate'] or args['run']:
        if args['--output_type'] not in ['json', 'csv']:
                raise Exception("--output_type has to be 'json' or 'csv
 ↪'")
if args['genconfig'] or args['generate'] or args['run']:
        if projectname_re.search(args['<projectname>']) is not None:
                raise Exception("Invalid <projectname>")
return
```

### 4.6.3 scrapple.utils.config

Functions related to traversing the configuration file

scrapple.utils.config.**traverse_next**(*page*, *nextx*, *results*, *tabular_data_headers=[]*, *verbosity=0*)

Recursive generator to traverse through the next attribute and crawl through the links to be followed.

**Parameters**

- **page** – The current page being parsed
- **next** – The next attribute of the current scraping dict
- **results** – The current extracted content, stored in a dict

**Returns** The extracted content, through a generator

In the case of crawlers, the configuration file can be treated as a tree, with the anchor tag links extracted from the follow link selector as the child nodes. This level-wise representation of the crawler configuration file provides a clear picture of how the file should be parsed.

This recursive generator performs a depth-first traversal of the config file tree. It can be implemented through this code snippet :

```
for link in page.extract_links(next['follow_link']):
        r = results.copy()
        for attribute in next['scraping'].get('data'):
                if attribute['field'] != "":
                        r[attribute['field']] = \
                        link.extract_content(attribute['selector'],
                        attribute['attr'],
                        attribute['default'])
        if not next['scraping'].get('next'):
                yield r
        else:
                for next2 in next['scraping'].get('next'):
                        for result in traverse_next(link, next2, r):
                                yield result
```

scrapple.utils.config.**get_fields**(*config*)

Recursive generator that yields the field names in the config file

**Parameters** **config** – The configuration file that contains the specification of the extractor

**Returns** The field names in the config file, through a generator

get_fields() parses the configuration file through a recursive generator, yielding the field names encountered.

```
for data in config['scraping']['data']:
        if data['field'] != '':
                yield data['field']
if 'next' in config['scraping']:
        for n in config['scraping']['next']:
                for f in get_fields(n):
                        yield f
```

scrapple.utils.config.**extract_fieldnames**(*config*)

Function to return a list of unique field names from the config file

**Parameters** **config** – The configuration file that contains the specification of the extractor

**Returns** A list of field names from the config file

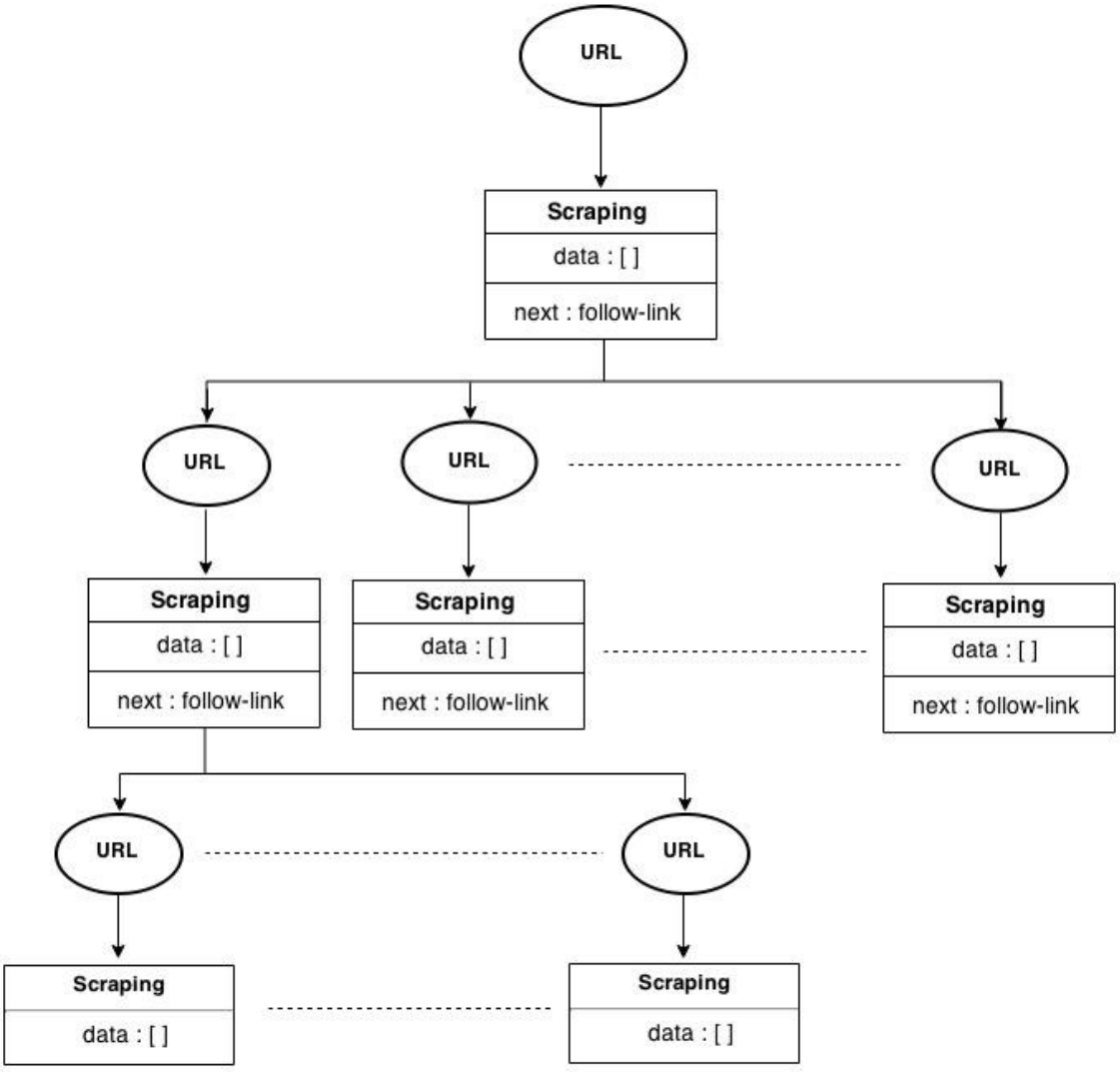


Fig. 4.6: Tree representation of crawler

The extract_fieldnames() function uses the get_fields() generator, and handles cases like multiple occurrences of the same field name.

```
fields = []
for x in get_fields(config):
        if x in fields:
                fields.append(x + '_' + str(fields.count(x) + 1))
        else:
                fields.append(x)
return fields
```

### 4.6.4 scrapple.utils.form

Functions related to form handling.

scrapple.utils.form.**form_to_json**(*form*)

Takes the form from the POST request in the web interface, and generates the JSON config file

**Parameters** **form** – The form from the POST request

**Returns** None

The web form is structured in a way that all the data field are linearly numbered. This is done so that it is easier to process the form while converting it into a JSON document.

```
for i in itertools.count(start=1):
        try:
                data = {
                        'field': form['field_' + str(i)],
                        'selector': form['selector_' + str(i)],
                        'attr': form['attribute_' + str(i)],
                        'default': form['default_' + str(i)]
                }
                config['scraping']['data'].append(data)
        except KeyError:
                break
```

***Scrapple implementation classes*** The classes involved in the implementation of Scrapple

***Interaction scenarios*** Interaction scenarios in the implementation of each of the Scrapple commands

***Command line interface*** The Scrapple command line interface

***Command classes*** The implementation of the command classes

***Selector classes*** The implementation of the selector classes

***Utility functions*** Utilities functions that support the implementation of the extractor

# CHAPTER 5

# Experimentation & Results

In this section, some experiments with Scrapple are provided. There are two main types of tools that can be implemented with the Scrapple framework :

- *Single page linear scrapers*
- *Link crawlers*

Once you've *installed Scrapple*, you can see the list of available *commands* and the related options using the command

```
$ scrapple --help
```

The *configuration file* is the backbone of Scrapple. It specifies the base page URL, selectors for the data extraction, the follow link for the link crawler and several other parameters.

Examples for each type are given.

## 5.1 Single page linear scrapers

For this example, we will extract content from a pyvideo page. Let us consider the following page.

To generate a skeleton configuration file, use the `genconfig` command. The primary arguments for the command are the project name and the URL of the base page.

```
$ scrapple genconfig pyvideo \
> http://pyvideo.org/video/1785/python-for-humans-1
```

This will create pyvideo.json which will initially look like this -

```
{

    "scraping": {
        "url": "http://pyvideo.org/video/1785/python-for-humans-1",
        "data": [
            {
                "field": "",
                "attr": "",
                "selector": "",
                "default": ""
            }
        ]
    },
    "project_name": "pyvideo",
    "selector_type": "xpath"

}
```

You can edit this json file to specify selectors for the various data that you would want to extract from the given page.

For example,

```
{

    "scraping": {
        "url": "http://pyvideo.org/video/1785/python-for-humans-1",
        "data": [
            {
                "field": "title",
                "attr": "text",
                "selector": "//h3",
                "default": ""
            },
            {
                "field": "speaker_name",
                "attr": "text",
                "selector": "//div[@id='sidebar']//dd[2]//a",
                "default": ""
            },
            {
                "field": "speaker_link",
                "attr": "href",
                "selector": "//div[@id='sidebar']//dd[2]//a",
                "default": ""
            },
```

```
            {
                "field": "event_name",
                "attr": "text",
                "selector": "//div[@id='sidebar']//dd[1]//a",
                "default": ""
            },
            {
                "field": "event_link",
                "attr": "href",
                "selector": "//div[@id='sidebar']//dd[1]//a",
                "default": ""
            }
        ]
    },
    "project_name": "pyvideo",
    "selector_type": "xpath"

}
```

Using this configuration file, you could generate a Python script using `scrapple generate` or directly run the scraper using `scrapple run`.

The `generate` and `run` commands take two positional arguments - the project name and the output file name.

To generate the Python script -

```
$ scrapple generate pyvideo talk1
```

This will create talk1.py, which is the script that can be run to replicate the action of `scrapple run`.

```
from __future__ import print_function
import json
import os

from scrapple.selectors.xpath import XpathSelector


def task_pyvideo():
        """
        Script generated using
        `Scrapple <http://scrappleapp.github.io/scrapple>`_
        """
        results = dict()
        results['project'] = "pyvideo"
        results['data'] = list()
        try:
```

```
                r0 = dict()
                page0 = XpathSelector(
                "http://pyvideo.org/video/1785/python-for-humans-1"
                )
                r0["title"] = page0.extract_content(
                "//h3", "text", ""
                )
                r0["speaker_name"] = page0.extract_content(
                "//div[@id='sidebar']//dd[2]//a", "text", ""
                )
                r0["speaker_link"] = page0.extract_content(
                "//div[@id='sidebar']//dd[2]//a", "href", ""
                )
                r0["event_name"] = page0.extract_content(
                "//div[@id='sidebar']//dd[1]//a", "text", ""
                )
                r0["event_link"] = page0.extract_content(
                "//div[@id='sidebar']//dd[1]//a", "href", ""
                )
                results['data'].append(r0)
        except KeyboardInterrupt:
                pass
        except Exception as e:
                print(e)
        finally:
                with open(os.path.join(os.getcwd(), 'talk1.json'), 'w
↪') as f:
                        json.dump(results, f)


if __name__ == '__main__':
        task_pyvideo()
```

To run the scraper -

```
$ scrapple run pyvideo talk1
```

This will create talk1.json, which contains the extracted information.

```
{

    "project": "test1",
    "data": [
        {
            "event_name": "PyCon US 2013",
            "event_link": "/category/33/pycon-us-2013",
            "speaker_link": "/speaker/726/kenneth-reitz",
```

```
            "speaker_name": "Kenneth Reitz",
            "title": "Python for Humans"
        }
    ]

}
```

## 5.2 Link crawlers

(Check out another example on the Github repo readme.)

For this example, we will extract content from all talks on pyvideo. We will use the event listing as the base page.

To generate a skeleton configuration file, use the `genconfig` command. The primary arguments for the command are the project name and the URL of the base page. To generate a skeleton configuration file for a crawler, use the `--type=crawler` argument.

```
$ scrapple genconfig pyvideo http://pyvideo.org/category \
> --type=crawler
```

This will create pyvideo.json which will initially look like this -

```
{

    "scraping": {
        "url": "http://pyvideo.org/category",
        "data": [
            {
                "default": "",
                "field": "",
                "attr": "",
                "selector": ""
            }
        ],
        "next": [
            {
                "follow_link": "",
                "scraping": {
                    "data": [
                        {
                            "default": "",
                            "field": "",
                            "attr": "",
                            "selector": ""
```

```
                        }
                    ]
                }
            }
        ]
    },
    "project_name": "pyvideo",
    "selector_type": "xpath"

}
```

You can edit this json file to specify selectors for the various data that you would want to extract from the given page.

For example,

```
{

    "scraping": {
        "url": "http://pyvideo.org/category/",
        "data": [
            {
                "field": "",
                "attr": "",
                "selector": "",
                "default": ""
            }
        ],
        "next": [
            {
                "follow_link": "//table//td[1]//a",
                "scraping": {
                    "data": [
                        {
                            "field": "event",
                            "attr": "text",
                            "selector": "//h1",
                            "default": ""
                        },
                        {
                            "field": "event_url",
                            "attr": "",
                            "selector": "url",
                            "default": ""
                        }
                    ],
                    "next": [
```

```
                        {
                            "follow_link": " \
                            //div[@id='video-summary-content']/div//
↪strong/a \
                            ",
                            "scraping": {
                                "data": [
                                    {
                                        "field": "talk_title",
                                        "attr": "text",
                                        "selector": "//h3",
                                        "default": "<unknown>"
                                    },
                                    {
                                        "field": "speaker",
                                        "attr": "text",
                                        "selector": " \
                                        //div[@id='sidebar']//dd[2] \
                                        ",
                                        "default": "<unknown>"
                                    },
                                    {
                                        "field": "talk_url",
                                        "attr": "",
                                        "selector": "url",
                                        "default": ""
                                    }
                                ]
                            }
                        }
                    ]
                }
            }
        ]
    },
    "project_name": "pyvideo",
    "selector_type": "xpath"

}
```

Using this configuration file, you could generate a Python script using `scrapple generate` or directly run the scraper using `scrapple run`.

The `generate` and `run` commands take two positional arguments - the project name and the output file name.

To generate the Python script -

```
$ scrapple generate pyvideo talk_list
```

This will create talk_list.py, which is the script that can be run to replicate the action of `scrapple run`.

```
from __future__ import print_function
import json
import os

from scrapple.selectors.xpath import XpathSelector


def task_pyvideo():
        """
        Script generated using
        `Scrapple <http://scrappleapp.github.io/scrapple>`_
        """
        results = dict()
        results['project'] = "pyvideo"
        results['data'] = list()
        try:
                r0 = dict()
                page0 = XpathSelector("http://pyvideo.org/category/")

                for page1 in page0.extract_links(
                "//table//td[1]//a"):
                        r1 = r0.copy()
                        r1["event"] = page1.extract_content(
                        "//h1", "text", ""
                        )
                        r1["event_url"] = page1.extract_content(
                        "url", "", ""
                        )


                for page2 in page1.extract_links(
                "//div[@class='video-summary-data']/div[1]//a"):
                        r2 = r1.copy()
                        r2["talk_title"] = page2.extract_content(
                        "//h3", "text", "<unknown>"
                        )
                        r2["speaker"] = page2.extract_content(
                        "//div[@id='sidebar']//dd[2]", "text", "
↪<unknown>"
                        )
                        r2["talk_url"] = page2.extract_content(
                        "url", "", ""
```

```
                        )
                        results['data'].append(r2)
        except KeyboardInterrupt:
                pass
        except Exception as e:
                print(e)
        finally:
                with open(os.path.join(os.getcwd(), 'talks.json'), 'w
↪') as f:
                        json.dump(results, f)


if __name__ == '__main__':
        task_pyvideo()
```

To run the scraper -

```
$ scrapple run pyvideo talk_list
```

This will create talk_list.json, which contains the extracted information.

A portion of the talk_list.json will look like this.


```
{

    "project": "pyvideo",
    "data": [
        {
            "talk_title": "Boston Python Meetup: ...",
            "talk_url": "http://pyvideo.org/video/591/...",
            "event_url": "http://pyvideo.org/category/15/...",
            "speaker": "Stephan Richter",
            "event": "Boston Python Meetup"
        },
        {
            "talk_title": "Boston Python Meetup: ...",
            "talk_url": "http://pyvideo.org/video/592/...",
            "event_url": "http://pyvideo.org/category/15/...",
            "speaker": "Marshall Weir",
            "event": "Boston Python Meetup"
        },
        {
            "talk_title": "November 2014 ...",
            "talk_url": "http://pyvideo.org/video/3359/...",
            "event_url": "http://pyvideo.org/category/14/...",
            "speaker": "Asma Mehjabeen Isaac Adorno",
            "event": "ChiPy"
```

```
        },


        ### talk_list.json continues


        {
            "talk_title": "Python 2.7 & Python 3: ...",
            "talk_url": "http://pyvideo.org/video/3373/...",
            "event_url": "http://pyvideo.org/category/64/...",
            "speaker": "Kenneth Reitz",
            "event": "Twitter University 2014"
        }
    ]

}
```

## 5.3 Comparison between Scrapple & Ducky

From the experiments performed with the Scrapple framework, we see that correctly written configuration files give accurate results. In the *single page linear extractor* example and *link crawler* example (where over 2800 pages were crawled through), an accuracy level of 100% was achieved.

The accuracy of the implementation of the Scrapple framework is dependent on the user's understanding of *web structure* and the ability to write correct *selector expressions*.

On comparison with Ducky [1], it can be seen that Ducky also provides an accuracy of 100%. The primary difference between the Scrapple framework and the Ducky framework is the features provided.

| Feature | Scrapple | Ducky |
|---|---|---|
| Configuration file | YES | YES |
| CSS selectors | YES | YES |
| XPath selectors | YES | NO |
| CSV output | YES | YES |
| JSON output | YES | YES |
| XML output | NO | YES |
| Generation of extractor script | YES | NO |

***Single page linear scrapers*** Tutorial for single page linear extractors

***Link crawlers*** Tutorial for link crawlers

CHAPTER 6

# Contributing to Scrapple

Scrapple is on GitHub !

## 6.1 Authors

Scrapple is written and maintained by the following contributors :

- Alex Mathew <alexmathew003@gmail.com>
- Harish Balakrishnan <harish.balakrishnan.93@gmail.com>

## 6.2 History

### 6.2.1 0.3.0 - 2016-09-23

- Set up table scraping parameters and execution
- Fix json configuration generation

### 6.2.2 0.2.6 - 2015-11-27

- Edit requirements

### 6.2.3 0.2.5 - 2015-05-28

- Add levels argument for genconfig command, to create crawler config files for a specific depth

### 6.2.4 0.2.4 - 2015-04-13

- Update documentation
- Minor fixes

### 6.2.5 0.2.3 - 2015-03-11

- Include implementation to use csv as the output format

### 6.2.6 0.2.2 - 2015-02-22

- Fix bug in generate script template

### 6.2.7 0.2.1 - 2015-02-21

- Update tests

### 6.2.8 0.2.0 - 2015-02-20

- Include implementation for `scrapple run` and `scrapple generate` for crawlers
- Modify web interface for editing scraper config files
- Revise skeleton configuration files

### 6.2.9 0.1.1 - 2015-02-10

- Release on PyPI with revisions
- Include web interface for editing scraper config files
- Modified implementations of certain functions

### 6.2.10 0.1.0 - 2015-02-04

- First release on PyPI

## 6.3 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

### 6.3.1 Types of Contributions

#### Report Bugs

Report bugs at https://github.com/scrappleapp/scrapple/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

#### Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to fix it.

#### Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

#### Write Documentation

Scrapple could always use more documentation, whether as part of the official Scrapple docs, in docstrings, or even on the web in blog posts, articles, and such.

#### Submit Feedback

The best way to send feedback is to file an issue at https://github.com/scrappleapp/scrapple/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

### 6.3.2 Get Started!

Ready to contribute? Here's how to set up *scrapple* for local development.

1. Fork the *scrapple* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/scrapple.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv scrapple
$ cd scrapple/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass nosetests:

```
$ cd tests && nosetests -v
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

### 6.3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/scrappleapp/scrapple/pull_requests and make sure that the tests pass for all supported Python versions.

***Authors*** The creators of Scrapple

***History*** History of Scrapple releases

***Contributing*** The Scrapple contribution guide

The goal of Scrapple is to provide a generalized solution to the problem of web content extraction. This framework requires a basic understanding of web page structure, which is necessary to write the necessary selector expressions. Using these selector expressions, the required web content extractors can be implemented to generate the desired datasets.

Experimentation with a wide range of websites gave consistently accurate results, in terms of the generated dataset. However, larger crawl jobs took a lot of time to complete and it was necessary to run the execution in one stretch. Scrapple could be improved to provide restartable crawlers, using caching mechanisms to keep track of the position in the URL frontier. Tag recommendation systems could also be implemented, using complex learning algorithms, though there would be a trade-off on accuracy.

# Python Module Index

## S

# Index

## T



## W



## X