DataHen Documentation

Perry Danoesubroto

Feb 19, 2020

Table of Contents

1 DataHen **1**
1.1 Getting Started 1

2 High Level Concepts **9**
2.1 Scrapers 11
2.2 Global Pages 11
2.3 Jobs 12
2.4 Job Workers 13
2.5 Job Pages 13
2.6 Job Outputs 14
2.7 Job Stats 15
2.8 Job Error Logs 15
2.9 Parsers 16
2.10 Seeder 17
2.11 Finisher 17
2.12 Exporters 18
2.13 Schemas 22

3 User Access **25**

4 Scraper Development workflow **27**

5 Scraper Maintenance workflow **29**

6 Coding Tutorials **31**
6.1 Simple Ebay scraper (Ruby) 31

7 Advanced Tutorials **65**
7.1 How to write a QA script to ingest and parse outputs from multiple scrapers 65

8 How-Tos **75**
8.1 Setting a scraper's scheduler 75
8.2 Changing a Scraper's or a Job's Proxy Type 76
8.3 Setting a specific ruby version 77
8.4 Setting a specific Ruby Gem 77
8.5 Changing a Scraper's Standard worker count 77
8.6 Changing a Scraper's Browser worker count 77

8.7 Changing an existing scrape job's worker count . . . 78
8.8 Enqueueing a page to Browser Fetcher's queue . . . 78
8.9 Setting fetch priority to a Job Page . . . 78
8.10 Setting a user-agent-type of a Job Page . . . 79
8.11 Setting the request method of a Job Page . . . 79
8.12 Setting the request headers of a Job Page . . . 79
8.13 Setting the request body of a Job Page . . . 79
8.14 Setting the page_type of a Job Page . . . 80
8.15 Reset a Job Page . . . 80
8.16 Handling cookies . . . 80
8.17 Force Fetching a specific unfresh page . . . 81
8.18 Handling JavaScript . . . 81
8.19 Browser display . . . 82
8.20 Browser interaction . . . 82
8.21 Taking screenshots . . . 84
8.22 Doing dry-run of your script locally . . . 87
8.23 Executing your script locally, and uploading to DataHen . . . 87
8.24 Querying scraper outputs . . . 87
8.25 Restart a scraping job . . . 88
8.26 Setting Environment Variables and Secrets on your account. . . . 88
8.27 Setting Input Variables and Secrets on your scraper and scrape job. . . . 89
8.28 Using a custom docker image for the scraper . . . 90
8.29 How to use shared code libraries from other Git repositories using Git Submodule . . . 91
8.30 How to debug page fetch . . . 91
8.31 Advanced Usage . . . 93

CHAPTER 1

DataHen

DataHen is a platform where you can scrape data from the internet quickly and easily without incurring significant costs. We do this by allowing you to scrape data from the shared-cache of contents that DataHen has collectively downloaded from the Internet for other users to scrape. Your scraping from the cached-content does not necessarily prevent you from getting the freshest content. In fact, you can specify how fresh the contents you want to scrape are by specifying your freshness-type and also specifying that the content should be "force fetched" or not.

Keep in mind that any page that DataHen downloads for your scraper, will be stored on the shared-cache, and will be available to other DataHen users to use for their own scrapers. This reuse of cached pages allows every DataHen users to collectively save cost and save time as they scrape data from the Internet. DataHen is like curl, where you have lower level control of HTTP request, such as request method, headers, body, and more.

1.1 Getting Started

In this getting started section, we will get you started with installing the necessary requirements, and then deploying and running an existing scraper into DataHen. Currently we support ruby 2.4.4 and 2.5.3.

1.1.1 Install DataHen Command Line Interface using rubygems

```
$ gem install datahen
Successfully installed datahen-0.2.3
Parsing documentation for datahen-0.2.3
Done installing documentation for datahen after 0 seconds
1 gem installed
```

1.1.2 Get your access token

You can create "account_admin" or "basic" token. The difference between the two is an "account_admin" token can create other access tokens, whereas basic account could not.

1.1.3 Set environment variable of your access token

```
$ export DATAHEN_TOKEN=<your_token_Here>
```

Now you're ready to go.

1.1.4 Create the scraper

In this step we will create a scraper on DataHen, by specifying the scraper name, and the git repository where the scraper script comes from:

```
$ hen scraper create walmart-movies git@git.datahen.com:scrapers/walmart-movies.git --
↪workers 1 --browsers 1
{
 "name": "walmart-movies",
 "id": 54,
 "account_id": 1,
 "force_fetch": false,
 "freshness_type": "any",
 "created_at": "2019-03-12T10:28:22.037768Z",
 "git_repository": "git@git.datahen.com:scrapers/walmart-movies.git",
 "git_branch": "master",
 "deployed_git_repository": "git@git.datahen.com:scrapers/walmart-movies.git",
 "deployed_git_branch": "master",
 "deployed_commit_hash": "e7d77d7622e7b71c32300eafd2d44a8429142fe3",
 "deployed_at": "2019-03-12T10:28:22.037768Z",
 "worker_count": 1,
 "config": {
  "parsers": [
   {
    "file": "./parsers/part.rb",
    "page_type": "part"
   }
  ],
  "seeder": {
   "file": "./seeder/seeder.rb"
  }
 }
}
```

Let's see if your scraper has been created. Let's look at the list of scrapers that you have now:

```
$ hen scraper list
[
 {
  "name": "ebay",
  "id": 20,
  "account_id": 1,
  "force_fetch": false,
  "freshness_type": "any",
  "created_at": "2018-11-26T22:00:43.007755Z",
  "git_repository": "https://github.com/DataHenOfficial/ebay-scraper.git",
  "git_branch": "master",
  "deployed_git_repository": "https://github.com/DataHenOfficial/ebay-scraper.git",
  "deployed_git_branch": "master",
  "deployed_commit_hash": "7bd6091d97a17cf8ee769e00ac285123c41aaf4f",
```

(continues on next page)

(continued from previous page)

```
  "deployed_at": "2018-11-28T06:13:56.571052Z",
  "worker_count": 1,
...
```

Or if you'd like to see your specific scraper, you can do:

```
$ hen scraper show walmart-movies
{
 "name": "walmart-movies",
 "id": 18,
 "account_id": 1,
 "force_fetch": false,
 "freshness_type": "any",
 "created_at": "2019-03-12T10:28:22.037768Z",
 "git_repository": "git@git.datahen.com:scrapers/walmart-movies.git",
 "git_branch": "master",
 "deployed_git_repository": "git@git.datahen.com:scrapers/walmart-movies.git",
...
```

Now that we have created the scraper, we need to deploy.

1.1.5 Deploying the scraper

Once we have created the scraper, let's deploy it from the git repo that you have specified.

```
$ hen scraper deploy walmart-movies
Deploying scraper. This may take a while...
{
 "id": 135,
 "scraper_id": 18,
 "commit_hash": "e7d77d7622e7b71c32300eafd2d44a8429142fe3",
 "git_repository": "git@git.datahen.com:scrapers/walmart-movies.git",
 "git_branch": "master",
 "errors": null,
 "success": true,
 "created_at": "2019-03-12T10:48:22.037768Z",
 "config": {
  "parsers": [
   {
    "file": "./parsers/part.rb",
    "page_type": "part"
   }
  ],
  "seeder": {
   "file": "./seeder/seeder.rb"
  }
 }
}
```

Let's see if the list of deployments, if you're curious to know your deployment history.

```
$ hen scraper deployment list walmart-movies
[
 {
  "id": 135,
```

(continues on next page)

(continued from previous page)

```
  "scraper_id": 18,
  "commit_hash": "e7d77d7622e7b71c32300eafd2d44a8429142fe3",
  "git_repository": "git@git.datahen.com:scrapers/walmart-movies.git",
  "git_branch": "master",
...
```

1.1.6 Run the scraper

Now that the scraper codes has been deployed, let's run it.

```
$ hen scraper start walmart-movies
Starting a scrape job...
{
 "id": 135,
 "scraper_id": 18,
 "created_at": "2019-03-12T10:52:22.037768Z",
 "freshness": null,
 "force_fetch": false,
 "status": "active",
 "seeding_at": null,
 "seeding_failed_at": null,
 "seeded_at": null,
 "seeding_try_count": 0,
 "seeding_fail_count": 0,
 "seeding_error_count": 0,
 "worker_count": 1
}
```

This will now then create a scraping job, which will start fetching pages for you, and parsing them into the outputs.

You can also see all jobs that was created on the scraper.

```
$ hen scraper job list walmart-movies
[
 {
  "id": 135,
  "scraper_name": "walmart-movies",
  "scraper_id": 18,
  "created_at": "2019-03-12T10:48:22.037768Z",
...
```

To view the current job on the scraper.

```
$ hen scraper job show walmart-movies
{
 "id": 135,
 "scraper_name": "walmart-movies",
 "scraper_id": 18,
 "created_at": "2019-03-12T10:48:22.037768Z",
...
```

1.1.7 Viewing the Job Stats

While the job is running, let's look how the job is doing by looking at the stats. You'll first need to get the ID form the job list command above.

```
$ hen scraper stats walmart-movies
{
 "job_id": 135,
 "pages": 822,
 "fetched_pages": 822,
 "to_fetch": 0,
 "fetching_failed": 0,
 "fetched_from_web": 0,
 "fetched_from_cache": 822,
 "parsed_pages": 0,
 "to_parse": 822,
 "parsing_failed": 0,
 "outputs": 0,
 "output_collections": 0,
 "workers": 1,
 "time_stamp": "2019-03-12T10:48:22.037768Z"
}
```

1.1.8 Viewing the Job Pages

Let's see the pages that has been added by the seeder script into this job.

```
$ hen scraper page list walmart-movies
[
 {
  "gid": "www.walmart.com-4aa9b6bd1f2717409c22d58c4870471e", # Global ID
  "job_id": 135,
  "page_type": "listings",
  "method": "GET",
  "url": "https://www.walmart.com/browse/movies-tv-shows/4096?facet=new_
↪releases:Last+90+Days",
  "effective_url": "https://www.walmart.com/browse/movies-tv-shows/4096?facet=new_
↪releases:Last+90+Days",
  "headers": "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/
↪537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
...
```

1.1.9 Viewing a Global Page Content

You may be wondering what is a Global Page. A Global Page acts like a shared-cache that DataHen fetches for all their users as they perform scraping. This shared-cache allows every users to collectively benefit from lower cost and higher performance of extracting data from the Internet.

Now that you've seen the pages that has been added into this job, let's see the content of the page by copying and pasting a page's GID(Global ID) into the following command.

```
$ hen globalpage content www.walmart.com-4aa9b6bd1f2717409c22d58c4870471e
Preview content url: "https://fetch.datahen.com/public/global_pages/preview/
↪HS2RNNi0uKe2YQ3tlU-
↪cedGCWhRHgLcm5PWTwTVx0VLs5yjlOt6bE8qma7lzv6oCfUSYBNHu3IpXK70961lRhcqruPg5xa29OmuSJvolz_
↪ONcVV2nmeMfJx8tSe_jRi8JW1qIfD7O8Rchf3XdO10pfjgICiV_
↪FBczWPGYmg3rNLGcHMk5UGseJcl7maAGvN5bhvrwesscrODp_mni894gKz8a9v3GTFtjVGUgexS-
↪dEu2DKTfe6SNb1ZKHj08SUCTM61P_Umg6XzF-bJBePMZuoX2b8nkXQ3mDw1-bdMJ-
↪wPF0fQ01T5gtkOCBDuSFBg-T8YGETNEPNm0usglfWzsq4="
```

(continues on next page)

(continued from previous page)

1.1.10 View the scraper output

Job Outputs are stored in collections. If none is specified, it will be stored in the "default" collection. Let's view the outputs of a scraper job by first seeing what collections the scraper outputs to:

```
$ hen scraper output collection walmart-movies
[
 {
  "collection": "products",
  "count": 72
 }
]
```

In the result of the command line above, you will see the collection called "products." Let's look at the outputs inside the "products" collection:

```
$ hen scraper output list walmart-movies --collection products
[
 {
  "_collection": "products",
  "_created_at": "2019-03-12T10:50:44.037768Z",
  "_gid": "www.walmart.com-a2232af59a8d52c356136f6674f532c5",
  "_id": "3de2e6b6e16749879f7e9bdd1ea3f0fc",
  "_job_id": 1341,
  "categories": [
   "Movies & TV Shows",
   "Movies",
   "Documentaries",
   "All Documentaries"
  ],
  "current_price": 21.89,
  "img_url": "https://i5.walmartimages.com/asr/5064efdd-9c84-4f17-a107-2669a34b54ff_1.
↪474fdc2d2d1ea64e45def9c0c5afb4c0.jpeg",
  "original_price": null,
  "publisher": "Kino Lorber",
  "rating": null,
  "reviews_count": 0,
  "title": "International Sweethearts of Rhythm (DVD)",
  "walmart_number": "572439718"
 },
...
```

1.1.11 View the scraper logs

If there is an error that occured it will be shown in the job log. Let's see what's in the log.

```
$ hen scraper log walmart-movies
```

You can view the log of what happens.

Congratulations! You've created and ran your first scraper.

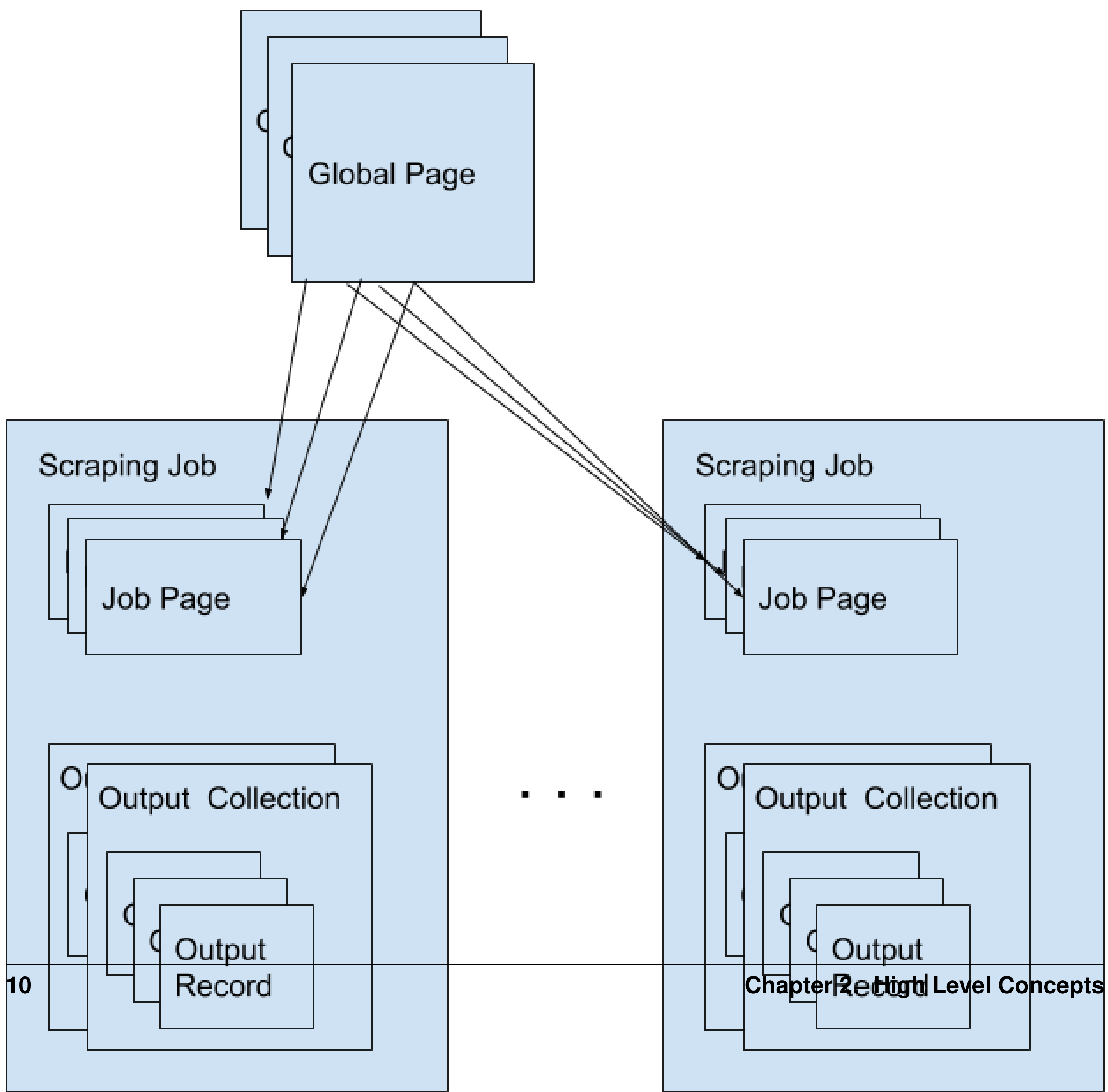
Let's now cleanup from this Getting Started section by canceling that running job.

```
$ hen scraper job cancel walmart-movies
{
 "id": 135,
 "scraper_name": "walmart-movies",
 "scraper_id": 18,
 "created_at": "2019-03-12T10:48:22.058468Z",
 "freshness": null,
 "force_fetch": false,
 "status": "cancelled",
 "seeding_at": "2019-03-12T10:49:42.035968Z",
 "seeding_failed_at": null,
 "seeded_at": "2019-03-12T10:50:23.057768Z",
 "seeding_try_count": 1,
 "seeding_fail_count": 0,
 "seeding_error_count": 0,
 "worker_count": 1
}
```

You're now done with the Getting Started section. Next steps are to read the high level concepts, and do the tutorials.

CHAPTER 2

High Level Concepts



2.1 Scrapers

A scraper is a group of tasks that allows users to extract data from the internet. Scrapers consists of a Seeder and Parsers (Currently we only support Ruby language).

2.1.1 Available commands

```
$ hen scraper help
Commands:
  hen scraper create <scraper_name> <git_repository>  # Create a scraper
  hen scraper delete <scraper_name>                   # Delete a scraper and related␣
↪records
  hen scraper deploy <scraper_name>                   # Deploy a scraper
  hen scraper deployment SUBCOMMAND ...ARGS           # manage scrapers deployments
  hen scraper export SUBCOMMAND ...ARGS               # manage scraper's exports
  hen scraper exporter SUBCOMMAND ...ARGS             # manage scraper's exporters
  hen scraper finisher SUBCOMMAND ...ARGS             # manage scrapers finishers
  hen scraper help [COMMAND]                          # Describe subcommands or one␣
↪specific subcommand
  hen scraper history <scraper_name>                  # Get historic stats for a job
  hen scraper job SUBCOMMAND ...ARGS                  # manage scrapers jobs
  hen scraper list                                    # List scrapers
  hen scraper log <scraper_name>                      # List log entries related to a␣
↪scraper's current job
  hen scraper output SUBCOMMAND ...ARGS               # view scraper outputs
  hen scraper page SUBCOMMAND ...ARGS                 # manage pages on a job
  hen scraper show <scraper_name>                     # Show a scraper
  hen scraper start <scraper_name>                    # Creates a scraping job and␣
↪runs it
  hen scraper stats <scraper_name>                    # Get the current stat for a job
  hen scraper update <scraper_name>                   # Update a scraper
  hen scraper var SUBCOMMAND ...ARGS                  # for managing scraper's␣
↪variables
```

2.2 Global Pages

All web pages that has been fetched by DataHen on behalf of users are stored in a shared cache, called Global Pages. Any web pages that you need to scrape will re-use this global pages if they fit within your freshness-type. If they don't match your freshness-type, you can specifically force-fetch them from your scraper and job settings.

2.2.1 Available Commands

```
$ hen globalpage help
Commands:
  hen globalpage content <gid>          # Show content of a globalpage
  hen globalpage failedcontent <gid>    # Show failed content of a globalpage
  hen globalpage help [COMMAND]         # Describe subcommands or one specific␣
↪subcommand
  hen globalpage show <gid>             # Show a global page
```

2.3 Jobs

When a scraper is run, it creates a scraping job, and will execute the seeder script that you've specified. As pages gets fetched, it will get parsed by the parser script that is related to that page.

A job has one of following possible statuses:

Status	Description
active	Job is running
paused	User manually paused the job
cancelled	User manually cancelled the job, or it is cancelled because another scheduled job on the same scraper has been started.
done	Job is done when there are no more to_fetch or to_parse

2.3.1 Available Commands

```
$ hen scraper job help
scraper job commands:
  hen scraper job cancel <scraper_name>   # cancels a scraper's current job
  hen scraper job help [COMMAND]          # Describe subcommands or one specific␣
↪subcommand
  hen scraper job list <scraper_name>     # gets a list of jobs on a scraper
  hen scraper job pause <scraper_name>    # pauses a scraper's current job
  hen scraper job resume <scraper_name>   # resumes a scraper's current job
  hen scraper job show <scraper_name>     # Show a scraper's current job
  hen scraper job update <scraper_name>   # updates a scraper's current job
```

2.3.2 Paused Jobs

Sometimes you may find that the status of your job has changed to "paused" on its own. This is a result of your scraper not having any more pages to process because the remaining pages are either in the parsed or failed queue. Specifically, a job will pause if there are no more pages remaining in the following queues:

```
to_fetch
fetching
to_parse
parsing_started
parsing
```

To check if there are any pages in the failed queue you can use the following stats command.

```
hen scraper stats <scraper_name>
```

You should look at the following failed queue counters and if there are failed pages:

```
fetching_failed
fetching_dequeue_failed
parsing_failed
parsing_dequeue_failed
```

Next step is to fix those failed pages and resume your job. You can use the following commands to list those pages and find the failed ones:

```
hen scraper page list <scraper_name> --fetch-fail  # to list fetch failed pages
hen scraper page list <scraper_name> --parse-fail  # to list parse failed pages
```

Then, once you have updated your scraper to fix any issues, you can refetch or reparse these pages using these commands:

```
hen scraper page refetch <scraper_name> --gid <gid>       # refetch an specific page
hen scraper page refetch <scraper_name> --fetch-fail      # refetch all fetch failed␣
↪pages
hen scraper page refetch <scraper_name> --parse-fail      # refetch all parse failed␣
↪pages
hen scraper page refetch <scraper_name> --status <queue>  # refetch all pages by queue
hen scraper page reparse <scraper_name> --gid <gid>       # reparse an specific page
hen scraper page reparse <scraper_name> --parse-fail      # reparse all parse failed␣
↪pages
hen scraper page reparse <scraper_name> --status <queue>  # reparse all pages by queue
```

After resetting at least one page, you can resume the job:

```
hen scraper job resume <scraper_name>
```

2.4 Job Workers

Job workers are units of capacity that a job can run. A job needs at least one worker for it to run.

There are two kinds of workers:

- Standard Worker. This allows you to fetch using regular HTTP method.
- Browser Worker. This will fetch using a real browser, and will render and execute any javascripts that are available on the page.

Typically one worker can has the capacity to perform:

- Fetching and parsing of up to 100,000 fresh pages per month from the internet. *
- Fetching and parsing of up to 300,000 pages per month from the shared cache(global page contents). *
- This totals to about 400,000 parsed pages per month. *

* performance varies based on many factors, including: target server capacity, bandwidth, size of pages, etc.

Note: If you need your scraping results sooner, you can purchase more capacity by adding more workers to your account and assigning more workers to your scraper. When you have multiple unused workers on your account, you can choose to either run multiple scrape jobs at once, or you can assign multiple workers to a single scrape job

2.5 Job Pages

Any Pages that are added by your scraper so that DataHen can fetch them, are all contained within the job, these are called job pages.

ForceFetch, when set to true, will force a page to be re-fetched if it is not fresh, as determined by freshness-type(day, week, month, year, any) that you have set on the scraper. Note: ForceFetch only works on pages that already exist in the DataHen platform. It has no effect on pages that does not exist, therefore, it will fetch the pages regardless if you force them to or not.

Vars. A job page can have user-defined variables, that you can set when a page is enqueued. This vars can then be used by the parser to do as you wish

Treat a page like a curl HTTP request, where you are in control of lower level things, such as, request method, body, headers, etc.

The following JSON describes the available options that you can use when enqueueing any page to DataHen via a script:

2.5.1 Available Commands

```
$ hen scraper page help
scraper page commands:
  hen scraper page add <scraper_name> <url>          # Enqueues a page to a scraper's␣
↪current job
  hen scraper page help [COMMAND]                    # Describe subcommands or one␣
↪specific subcommand
  hen scraper page list <scraper_name>               # List Pages on a scraper's␣
↪current job
  hen scraper page log <scraper_name> <gid>          # List log entries related to a␣
↪job page
  hen scraper page refetch <scraper_name> <options>  # Refetch Pages on a scraper's␣
↪current job
  hen scraper page reparse <scraper_name> <options>  # Reparse Pages on a scraper's␣
↪current job
  hen scraper page show <scraper_name> <gid>         # Show a page in scraper's␣
↪current job
  hen scraper page update <scraper_name> <gid>       # Update a page in a scraper's␣
↪current job
```

2.6 Job Outputs

Outputs are generated by parser scripts. Outputs are contained within a collection that you can specify. By default, if you don't specify a collection, the output will be stored in the "default" collection. Job outputs are in JSON format.

2.6.1 Available Commands

```
$ hen scraper output help
scraper output commands:
  hen scraper output collections <scraper_name>       # list job output collections␣
↪that are inside a current job of a scraper.
  hen scraper output help [COMMAND]                   # Describe subcommands or one␣
↪specific subcommand
  hen scraper output list <scraper_name>              # List output records in a␣
↪collection that is in the current job
  hen scraper output show <scraper_name> <record_id>  # Show one output record in a␣
↪collection that is in the current job of a scraper
```

2.7 Job Stats

Knowing your job stats is important and being able to analyze your job stats over the time even more. Datahen understands this and keeps historic stats data on all your jobs for further analyze.

2.7.1 Available Commands

To check your job current stats you can use the following stats command.

```
hen scraper stats <scraper_name>
```

To check your job historic stats you can use the following history command.

```
$ hen scraper help history
Usage:
  hen scraper history <scraper_name>

Options:
  j, [--job=N]                          # Set a specific job ID
      [--min-timestamp=MIN-TIMESTAMP]   # Starting timestamp point in time to query␣
↪historic stats (inclusive)
      [--max-timestamp=MAX-TIMESTAMP]   # Ending timestamp point in time to query␣
↪historic stats (inclusive)
      [--limit=N]                       # Limit stats retrieved
      [--order=N]                       # Order stats by timestamp [DESC]

Description:
  Get historic stats for a scraper's current job
```

2.8 Job Error Logs

When an error occurs inside a job, it gets logged. And you can check to see the errors that occur on a job, or even on a particular page

2.8.1 Available Commands

```
$ hen scraper help log
Usage:
  hen scraper log <scraper_name>

Options:
  j, [--job=N]              # Set a specific job ID
  H, [--head=HEAD]          # Show the oldest log entries. If not set, newest entries␣
↪is shown
  p, [--parsing=PARSING]    # Show only log entries related to parsing errors
  s, [--seeding=SEEDING]    # Show only log entries related to seeding errors
  m, [--more=MORE]          # Show next set of log entries. Enter the `More token`
```

Description: Shows log related to a scraper's current job. Defaults to showing the most recent entries

```
$ hen scraper page help log
Usage:
  hen scraper page log <scraper_name> <gid>

Options:
  j, [--job=N]              # Set a specific job ID
  H, [--head=HEAD]          # Show the oldest log entries. If not set, newest entries␣
↪is shown
  p, [--parsing=PARSING]   # Show only log entries related to parsing
  m, [--more=MORE]          # Show next set of log entries. Enter the `More token`
```

Description: Shows log related to a page in the job. Defaults to showing the most recent entries

2.9 Parsers

Parsers are scripts that you create within a scraper in order to extract data from a web page, or to enqueue other pages. The parser scripts are executed as soon as a page is downloaded. You can create a script for a particular type of page, for example, if you were to scrape an e-commerce website, you can have an "index" page type, and a "detail" page type. When you enqueue a page to DataHen, you need to specify the page_type so that the matching parsers for that page_type will be executed.

2.9.1 Reserved words or methods in parser scripts:

```
page # => Hash. returns the page metadata
page['vars'] # => Hash. returns the page's user-defined variables
content # => String. returns the actual response body of the page
pages # => []. the pages to be enqueued, which will be fetched later
outputs # => []. the array of job output to be saved
save_pages(pages) # Save an array of pages right away and remove all elements from␣
↪the array. By default this is not necessary because the parser will save the "pages
↪" variable. However, if we are saving large number of pages (thousands), it is␣
↪better to use this method, to avoid storing everything in memory
save_outputs(outputs) # Save an array of outputs right away and remove all elements␣
↪from the array. By default this is not necessary because the parser will save the
↪"outputs" variable. However, if we are saving large number of outputs (thousands),␣
↪it is better to use this method, to avoid storing everything in memory
```

2.9.2 Available Commands

```
$ hen parser help
Commands:
  hen parser exec <scraper_name> <parser_file> <GID>...<GID>   # Executes a parser␣
↪script on one or more Job Pages within a scraper's current job
  hen parser help [COMMAND]                                    # Describe subcommands␣
↪or one specific subcommand
  hen parser try <scraper_name> <parser_file> <GID>            # Tries a parser on a␣
↪Job Page
```

2.10 Seeder

Seeder script is a script that is executed at the start of any job, that allows you to enqueue URLs that needs to be fetched by DataHen.

To Add a seeder, you simply add the following to your config.yaml file:

```
seeder:
  file: ./seeder/seeder.rb
  disabled: false
```

2.10.1 Reserved words or methods in seeder scripts:

```
pages # => []. The pages to be enqueued, and will be fetched later
outputs # => []. the array of job output to be saved
save_pages(pages) # Save an array of pages right away and remove all elements from␣
↪the array. By default this is not necessary because the seeder will save the "pages
↪" variable. However, if we are seeding large number of pages (thousands), it is␣
↪better to use this method, to avoid storing everything in memory
save_outputs(outputs) # Save an array of outputs right away and remove all elements␣
↪from the array. By default this is not necessary because the seeder will save the
↪"outputs" variable. However, if we are saving large number of outputs (thousands),␣
↪it is better to use this method, to avoid storing everything in memory
```

2.10.2 Available Commands

```
$ hen seeder help
Commands:
  hen seeder exec <scraper_name> <seeder_file>  # Executes a seeder script onto a␣
↪scraper's current job.
  hen seeder help [COMMAND]                     # Describe subcommands or one␣
↪specific subcommand
  hen seeder try <scraper_name> <seeder_file>                  # Tries a seeder file
```

2.11 Finisher

Finisher script is a script that is executed at the end of any job. This allows you to perform actions after your scraper job is done such as creating summaries and starting exporters.

To Add a finisher, you simply add the following to your config.yaml file:

```
finisher:
  file: ./finisher/finisher.rb
  disabled: false
```

2.11.1 Reserved words or methods in finisher scripts:

```
job_id # The id of the job that has just finished
outputs # => []. the array of job output to be saved
save_outputs(outputs) # Save an array of outputs right away and remove all elements␣
↪from the array. By default this is not necessary because the seeder will save the
↪"outputs" variable. However, if we are saving large number of outputs (thousands),␣
↪it is better to use this method, to avoid storing everything in memory
```

2.11.2 Available Commands

```
hen finisher help
Commands:
  hen finisher exec <scraper_name> <finisher_file>  # Executes a finisher script onto␣
↪a scraper's current job.
  hen finisher help [COMMAND]                        # Describe subcommands or one␣
↪specific subcommand
  hen finisher try <scraper_name> <finisher_file>   # Tries a finisher file

hen scraper finisher help
scraper finisher commands:
  hen scraper finisher help [COMMAND]        # Describe subcommands or one specific␣
↪subcommand
  hen scraper finisher reset <scraper_name>  # Reset finisher on a scraper's current␣
↪job
```

2.12 Exporters

Exporters are a set of configurations that allows you to export data from DataHen into various formats. We currently have several different exporters: JSON, CSV, and Content. To add an exporter, you simply just add some lines of code under your *exporters* section of your config.yaml like the following example:

```yaml
seeder:
...
parsers:
...
# the following lines define exporters...
exporters:
 - exporter_name: products_json_short # Example JSON Exporter
   exporter_type: json
   collection: products
   write_mode: line
   limit: 100
   offset: 10
 - exporter_name: details_content_short # Example Content Exporter
   exporter_type: content
   page_type: details
   limit: 100
   offset: 10
```

Once you have added the above configuration, you need to deploy the scraper first before you can start creating exports. IMPORTANT: Exporter Names must be unique per scraper, because this is how you're going to run the exporter with.

2.12.1 Available Exporter Commands

```
$ ae scraper exporter help
scraper exporter commands:
  hen scraper exporter list <scraper_name>
  hen scraper exporter show <scraper_name> <exporter_name>
  hen scraper exporter start <scraper_name> <exporter_name>
```

2.12.2 Available Export Commands

```
$ ae scraper export help
scraper export commands:
  hen scraper export download <export_id>
  hen scraper export list                    # Gets a list
  hen scraper export show <export_id>        # Show an export
```

2.12.3 Automatically Start Exporters

You can automatically start any exporter as soon as the scrape job is done. To do this, simply add `start_on_job_done: true` to your exporter configuration. The following is an example config file that has the exporters ready to auto-start.

```
seeder:
...
parsers:
...
# the following lines define exporters...
exporters:
 - exporter_name: products_json_short # Example JSON Exporter
   exporter_type: json
   collection: products
   write_mode: line
   limit: 100
   offset: 10
   start_on_job_done: true # This field will auto start this exporter
 - exporter_name: details_content_short # Example Content Exporter
   exporter_type: content
   page_type: details
   limit: 100
   offset: 10
   start_on_job_done: true # This field will auto start this exporter
```

2.12.4 JSON Exporter

Json exporter allows you to export a collection into json formatted file. Typically, a JSON Exporter looks like this:

```
exporter_name: <your_exporter_name_here> # Must be unique
exporter_type: json
collection: <collection_here>
write_mode: line # can be `line`,`pretty`, `pretty_array`, or `array`
limit: 100 # limits to how many records to export
```

(continues on next page)

(continued from previous page)

```
offset: 10
start_on_job_done: true
```

JSON Write Modes

The JSON exporter supports four different write modes, based on your needs: line, pretty, pretty_array, and array.

Write mode of *line* will export a file with the following content:

```
{"foo1": "bar1"}
{"foo1": "bar1"}
{"foo1": "bar1"}
```

Write mode of *pretty* will export a file with the following content:

```
{
  "foo1": "bar1"
}
{
  "foo1": "bar1"
}
{
  "foo1": "bar1"
}
```

Write mode of *pretty_array* will export the following content:

```
[{
  "foo1": "bar1"
},
{
  "foo1": "bar1"
},
{
  "foo1": "bar1"
}]
```

Write mode of *array* will export the following content:

```
[{"foo1": "bar1"},
{"foo1": "bar1"},
{"foo1": "bar1"}]
```

2.12.5 CSV Exporter

CSV exporter allows you to export a collection into a CSV formatted file. Typically, a CSV Exporter looks like this:

```
exporter_name: <your_exporter_name_here> # Must be unique
exporter_type: csv
collection: <collection_here>
no_headers: false # Specifies if you want the headers row. Default: false
limit: 100 # limits to how many records to export
start_on_job_done: true
fields:
```

(continues on next page)

(continued from previous page)

```
- header: "gid"
  path: "_gid"
- header: "some_value"
  path: "some_value"
- header: "some_nested_value"
  path: "path.to.your.value"
```

CSV Fields

Pay careful attention to this fields configuration, as, this is where you need to specify the header and the path, so that the CSV exporter knows how to write the csv rows. A CSV Field, contains two attributes, Header, and Path.

Header allows you to set the value of the csv header.

Path allows the CSV exporter to traverse your output record in order to find the correct value based on the dot "." deliminator. Take a look at the following output record:

```
{
  "foo1": "bar1",
  "foo2":  { "sub2" : "subvalue2" }
}
```

In the above example, the path "foo1" produces the value: "bar1" And the path "foo2.sub2" produces the value "subvalue2"

With this combination of Header and Path, the CSV exporter should cover a lot of your use cases when it comes to exporting CSVs. However, if you feel that you have a rare scenario where you're not able to traverse the output very well by using Path, you should code your parser scripts to output a simpler schema.

2.12.6 Content Exporter

Content exporter allows you to export the actual content of the page that has been fetched for you. You can export any contents including html, pdf, images, etc. The difference between Content exporter and other exporters, is that, it exports from the list of Pages that you have on your scraper job.

When the exported has done exporting, you will get the actual content files, as well as a CSV file that contains a list of all the contents that has been exported. You can use that CSV file, to know what content files has been exported. This is especially useful, if you want to ingest and process these content files in another system.

Typically, a Content Exporter looks like this:

```
exporter_name: <your_exporter_name_here> # Must be unique
exporter_type: content
page_type: <page_type>
filename_var: <filename_var> # variable to refer to, when naming the file
ignore_extensions: false # filename will have no extension, if true
include_failed_contents: false # self explanatory. Helpful for troubleshooting
limit: 100 # limits to how many records to export
start_on_job_done: true
```

Exporting Failed Contents

You can specify to export failed contents as well, this will come handy for troubleshooting purposes. On your exporter's config, set the following to true:

```
include_failed_contents: true
```

When you have specified this to be true, this exporter will save any failed contents in a separate directory.

Note: Keep in mind that failed contents are not saved as a file with their GID as their default filename. They are saved with their CID(Content ID) as the filename. The reason is to remove duplication, as most failed requests to websites display the same exact content repetitiously.

Customizing the File Names

By default, the Content exporter export the content files, with a standard naming convention of:

```
<gid>.<ext>
```

If you want to specify a name for the files, you need to set that in the page's variable, and tell our exporter about what variable it should be. For example, let's say you have the following Page

```
{
  "gid": "www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e",
  "url": "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682",
}
```

By default, this will export the page content and save it with the following filename:

```
www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e.html
```

Let's say you want this file to be saved with this filename:

```
9335.html
```

You would need to enqueue that page with a variable, like so:

```
pages << {
    url: "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682",
    vars: {
        my_filename: "9335", # notice we added a "my_filename" var
    }
  }
```

And then we would need to set the exporter's filename_var config like the following:

```
exporter_name: <your_exporter_name_here>
exporter_type: content
page_type: <page_type>
filename_var: my_filename # Need to tell the exporter how to name the file
```

And that's it. This particular content will be then saved as a file with the following filename:

```
9335.html
```

2.13 Schemas

For output records that needs to follow a certain schema, we support the use of json-schema.org v4, v6, and v7 to validate your collection outputs.

To learn more on how to write your schema files, please visit Understanding JSON Schema.

You can also easily generate a your JSON schema, from a regular JSON record by visiting: jsonschema.net. Doing so will make it much easier to get started with building your schema files.

To see an example of how a scraper uses a schema, visit the following project.

To specify any schema to collection(s), you need to do the following steps:

2.13.1 1. Create the json schema file

Ideally the convention to organize your schema files is to create a directory called `./schemas` in the root project directory, and then put all the related files inside. In this example let's create a schema file that will validate contact information. In this case, you can create the file `./schemas/contact.json` with the following content:

```
{
  "type": "object",
  "properties": {
    "name":      { "type": "string" },
    "email":     { "type": "string" },
    "address":   { "type": "string" },
    "telephone": { "type": "string" }
  },
  "required": ["name", "email"]
}
```

This file contains the actual json-schema that will be used to validate an output record.

2.13.2 2. Create the schema config file and list the schema file that will be used to validate the collection(s)

Once you've created the schema file in step 1, you now need to create a schema config file. Let's create the file `./schemas/config.yaml` file with the following content:

```
schemas:
 - file: ./schemas/contacts.json
   collections: "contacts,contacts1,contacts2" # you can put multiple collections to␣
↪be validated by the same schema file
   disabled: false
```

2.13.3 3. Update your config.yaml file to include the schema config file.

Once you've created the schema config file, you now need to refer to this schema config file from your project's main config YAML file. Now, add the following content to your `./config.yaml`

```
schema_config:
  file: ./schemas/config.yaml
  disabled: false
```

Once this is done, and you've deployed your scraper, any time your script will try to save any output into your specified collections, they will be validated based on the schemas that you've specified.

CHAPTER 3

User Access

A user with account_admin role can issue access token to others and set roles, so that others can help them create and maintain scrapers.

CHAPTER 4

Scraper Development workflow

1. Create a scraper git repository
2. Create the scraper on DataHen
3. Write a seeder and parser scripts
4. Run a seeder script locally against DataHen to see if it works
5. Run the parser scripts locally against the global pages
6. Deploy the scraper
7. Run the scraper on DataHen
8. Check the job outputs on DataHen

CHAPTER 5

Scraper Maintenance workflow

1. Look at the job outputs and/or job logs
2. If there is an error, pause or cancel the affected job on the DataHen scraper
3. Go to the scraper directory locally
4. Modify the seeder script or parser scripts locally
5. Run the seeder script or parser script locally against some fetched job pages
6. Deploy the scraper
7. Resume the scrape job, or run another scrape by creating another scrape job.
8. Check the job outputs on DataHen

CHAPTER 6

Coding Tutorials

6.1 Simple Ebay scraper (Ruby)

Let's create an ebay scraper from scratch. In this tutorial, we will go step by step in creating a scraper that scrapes an ebay listing page, and loop through each of the pages to extract some related information from it. You will also get familiarized with the basic usage of developing on the DataHen platform.

If you want to see the the exercise solutions, please refer to the branches in https://github.com/DataHenOfficial/ebay-scraper

6.1.1 Exercise 1: Create a base scraper locally

In this exercise we will create a base scraper. At minimum a scraper needs a seeder script for it to run. After that we will commit and deploy to DataHen.

Let's create an empty directory first, and name it 'ebay-scraper'

```
$ mkdir ebay-scraper
```

Next let's go into the directory and initialize it as a Git repository

```
$ cd ebay-scraper
$ git init .
Initialized empty Git repository in /Users/workspace/ebay-scraper/.git/
```

Now, let's create a seeder script. A seeder script gets run when a scraping job is run. This allows for us to seed the first few pages that will need to be fetched and parsed by DataHen.

Let's create a seeder directory first, so that we can put any files related to our seeding there. This is a good convention to keep our codes organized

```
$ mkdir seeder
```

Next, let's create a ruby script called seeder.rb in the seeder directory with the following content:

```
pages << {
  page_type: 'listings',
  method: "GET",
  url: "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682"
}
```

In the ruby script above, we are basically seeding an Apple iPhone page on ebay, so that DataHen can download the page so that it can be parsed. We are also setting page_type to 'listings', this allows DataHen to know which parser script to use to parse the content of this page. More on this on Exercise 3. Note that *pages* is a reserved variable. It is an array that represents what pages you want to seed, so that it gets fetched.

After you've created the seeder.rb file, the seeder directory, should look like this:

```
$ ls -alth
total 8
-rw-r--r--  1 johndoe  staff   122B 26 Nov 16:16 seeder.rb
drwxr-xr-x  3 johndoe  staff    96B 26 Nov 16:15 .
drwxr-xr-x  4 johndoe  staff   128B 26 Nov 16:15 ..
```

Let's go back to the root directory of the project

```
$ cd ..
$ ls -alth
total 0
drwxr-xr-x  10 johndoe  staff   320B 26 Nov 16:19 .git
drwxr-xr-x   3 johndoe  staff    96B 26 Nov 16:15 seeder
drwxr-xr-x   4 johndoe  staff   128B 26 Nov 16:15 .
drwxr-xr-x  10 johndoe  staff   320B 26 Nov 15:59 ..
```

Now that we've created the seeder script, let's see if there are any syntax error in it. Luckily for us, by using the datahen command line interface(cli), we can trial-run a script without actually saving it to DataHen.

From the root of project directory, let's try the seeder script.

```
$ hen seeder try ebay seeder/seeder.rb
Trying seeder script
=========== Seeding Script Executed ===========
----------- New Pages to Enqueue: -----------
[
  {
    "page_type": "listings",
    "method": "GET",
    "url": "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682"
  }
]
```

If you see the above, it means the trial-run was successful.

Let's now commit this git repository

```
$ git add .
$ git commit -m 'created a seeder file'
[master (root-commit) 7632be0] created a seeder file
 1 file changed, 5 insertions(+)
 create mode 100644 seeder/seeder.rb
```

Next, let's push it to a online git repository provider. In this case let's push this to github. In the example below it is using our git repository, you should push to your own repository.

```
$ git remote add origin https://github.com/DataHenOfficial/ebay-scraper.git
$ git push -u origin master
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 382 bytes | 382.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/DataHenOfficial/ebay-scraper/pull/new/master
remote:
To https://github.com/DataHenOfficial/ebay-scraper.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Ok, looks like this has successfully pushed.

Let's now create a config file. A scraper requires a config file so that DataHen understands how to seed, and do other things. Create a config.yaml file in the root project directory with the following content:

```
seeder:
 file: ./seeder/seeder.rb
 disabled: false # Optional. Set it to true if you want to disable execution of this
↪file
```

The config above simply tells DataHen where the seeder file is, so that it can be executed.

Let's now commit this config file on git, and push it to Github.

```
$ git add .
$ git commit -m 'add config.yaml file'
[master c32d451] add config.yaml file
 1 file changed, 3 insertions(+)
 create mode 100644 config.yaml
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 383 bytes | 383.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/DataHenOfficial/ebay-scraper.git
   7632be0..c32d451  master -> master
```

Congratulations, you've successfully created a base scraper that includes a seeder script and a config file. You've also pushed this scraper codes to Github.

In the next exercise we'll learn how to run this scraper on DataHen.

6.1.2 Exercise 2: Run the scraper on DataHen

In the last exercise, you've learned to create a bare minimum requirement of a scraper. Let's now run this scraper on DataHen. If you've skipped the last exercise, you can see the source code here: https://github.com/DataHenOfficial/ebay-scraper/tree/exercise1

Let's run the code on DataHen now. But before we do that, we need to create the scraper first using the code that was created. You also need the URL to your scraper's Git repository.

Create the scraper on DataHen and name it 'ebay.'

```
$ hen scraper create ebay https://github.com/DataHenOfficial/ebay-scraper.git
{
 "name": "ebay",
 "id": 20,
 "account_id": 1,
 "force_fetch": false,
 "freshness_type": "any",
 "created_at": "2018-11-26T22:00:43.007755Z",
 "git_repository": "https://github.com/DataHenOfficial/ebay-scraper.git",
 "git_branch": "master",
 "deployed_git_repository": null,
 "deployed_git_branch": null,
 "deployed_commit_hash": null,
 "deployed_at": null,
 "config": null
}
```

If you see the above, that means it successfully created the scraper.

Next, we need to deploy from your remote Git repository onto DataHen.

```
$ hen scraper deploy ebay
Deploying scraper. This may take a while...
{
 "id": 122,
 "scraper_id": 20,
 "commit_hash": "c32d4513dbe3aa8441fa6b80f3ffcc5d84fb7a03",
 "git_repository": "https://github.com/DataHenOfficial/ebay-scraper.git",
 "git_branch": "master",
 "errors": null,
 "success": true,
 "created_at": "2018-11-26T22:03:09.002231Z",
 "config": {
  "seeder": {
   "file": "./seeder/seeder.rb"
  }
 }
}
```

Seems like the deployment was a success, and that there were no errors.

You can also see the deployment history of this scraper as well.

```
$ hen scraper deployment list ebay
[
 {
  "id": 122,
  "scraper_id": 20,
  "commit_hash": "c32d4513dbe3aa8441fa6b80f3ffcc5d84fb7a03",
  "git_repository": "https://github.com/DataHenOfficial/ebay-scraper.git",
  "git_branch": "master",
  "errors": null,
  "success": true,
  "created_at": "2018-11-26T22:03:09.002231Z",
  "config": {
   "seeder": {
    "file": "./seeder/seeder.rb"
   }
```

(continues on next page)

(continued from previous page)

```
    }
  }
]
```

Of course, because there was only one deployment, you only see one here.

Let's now start the scraper.

```
$ hen scraper start ebay
Starting a scrape job...
{
 "id": 70,
 "scraper_id": 20,
 "created_at": "2018-11-26T22:06:54.399547Z",
 "freshness": null,
 "force_fetch": false,
 "status": "active",
 "seeding_at": null,
 "seeding_failed_at": null,
 "seeded_at": null,
 "seeding_try_count": 0,
 "seeding_fail_count": 0,
 "seeding_error_count": 0,
 "worker_count": 1
}
```

Doing the above will create a new scrape job and run it. Notice that the job status is "active". This means that the scraper job is currently running.

Let's give it a minute, and see the stats of the job:

```
$ hen scraper stats ebay
{
 "job_id": 70,                 # Job ID
 "pages": 1,                   # How many pages in the scrape job
 "fetched_pages": 1,           # Number of fetched pages
 "to_fetch": 0,                # Pages that needs to be fetched
 "fetching_failed": 0,         # Pages that failed fetching
 "fetched_from_web": 1,        # Pages that were fetched from Web
 "fetched_from_cache": 0,      # Pages that were fetched from the shared Cache
 "parsed_pages": 0,            # Pages that have been parsed by parsing script
 "to_parse": 1,                # Pages that needs to be parsed
 "parsing_failed": 0,          # Pages that failed parsing
 "outputs": 0,                 # Outputs of the scrape
 "output_collections": 0,      # Output collections
 "workers": 1,                 # How many workers are used in this scrape job
 "time_stamp": "2018-11-26T22:09:57.956158Z"
}
```

From the stats above, we can derive that one page has been seeded, and that one page has been fetched by our scrape job. So this looks Good.

Just to be sure, let's check the scraper log to see if there are any errors:

```
$ hen scraper log ebay
```

Seems like there are no errors so far.

You have now successfully ran a scrape job on DataHen.

Congratulations, you have completed exercise 2.

To see the codes that was done throughout this exercise, please visit https://github.com/DataHenOfficial/ebay-scraper/tree/exercise2

In the next exercise, we'll learn how to write a parser script so that we can parse the pages that has been enqueued by the seeder.

6.1.3 Exercise 3. Create parser script

In this exercise, we'll learn how to write a parser script so that we can parse the pages that has been fetched, and output it into a collection. If you have not done Exercise 2, please do so first, as this exercise depends on exercise 2.

To continue where we left off, let's look at the pages that has been enqueued and fetched.

Look at the page list that is in the ebay scraper.

```
$ hen scraper page list ebay
[
 {
  "gid": "www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e", # Global ID
  "job_id": 70,
  "page_type": "listings",
  "method": "GET",
  "url": "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682",
  "effective_url": "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682",
  "headers": null,
  "body": null,
  "created_at": "2018-11-26T22:07:49.013537Z",
  ...
  "fetched_at": "2018-11-26T22:08:03.14285Z",
  ...
 }
]
```

This returns the page that the seeder have enqueued. Also, take note of the GID field **www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e** GID (Global ID) is a very important concept in DataHen. This represents a unique identifier of that particular page that you've enqueued. You can refer to this particular page by referring to this GID.

If you want to see the scraper job's page in, you can do like so:

```
$ hen scraper page show ebay www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e
{
 "gid": "www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e",
 "job_id": 70,
 "page_type": "listings",
 "method": "GET",
 "url": "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682",
 "effective_url": "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682",
...
```

You can also take a look at the Global page for this GID. Think of the global page as a shared-cache of metadatas and contents related to pages that all DataHen users have fetched. Globalpage contains many more information about the page than the scraper's job page. Please refer to the "High Level Concepts" section to learn more.

```
$ hen globalpage show www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e
{
```

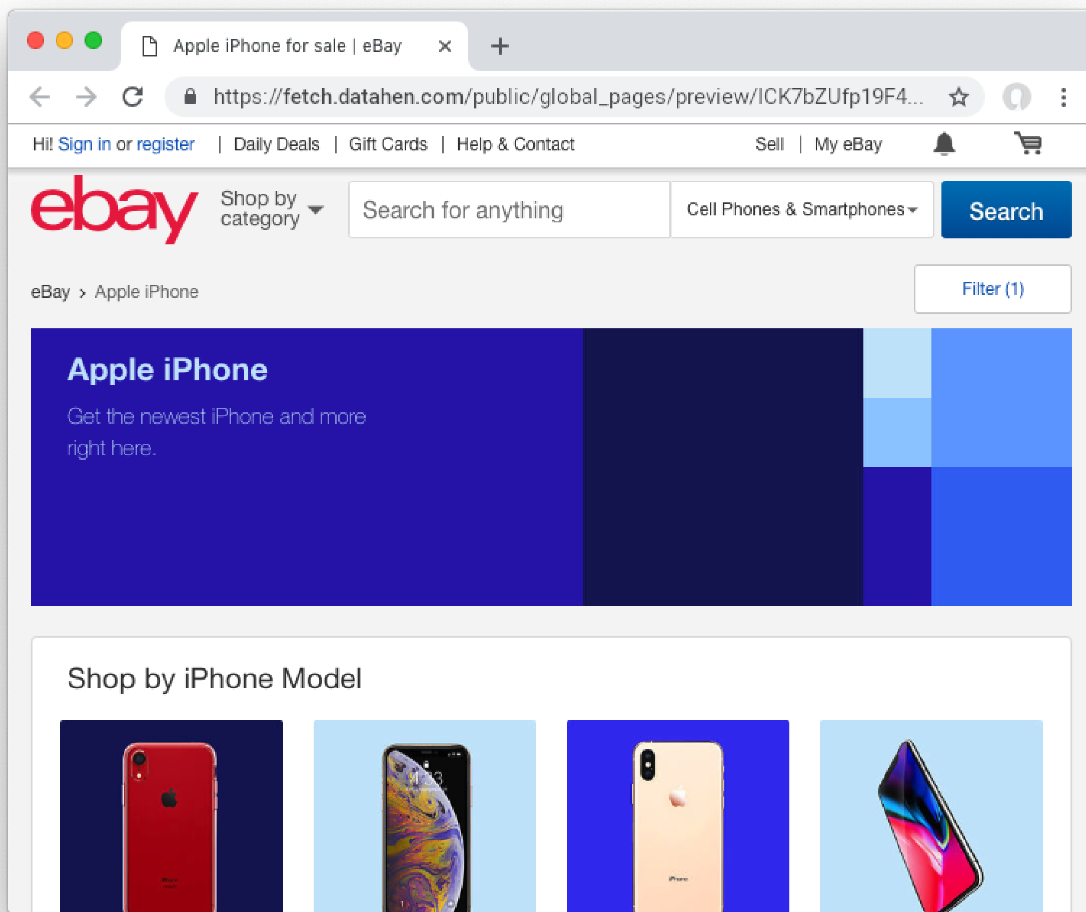
(continues on next page)

(continued from previous page)

```
  "gid": "www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e",
  "hostname": "www.ebay.com",
  "method": "GET",
  "url": "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682",
  "effective_url": "https://www.ebay.com/b/Apple-iPhone/9355/bn_319682",
  "headers": null,
  "body": null,
...
```

Now that you've seen an information of a global page, let's look at the actual content that is stored in the cache. Let's take the same GID and preview the content of that page

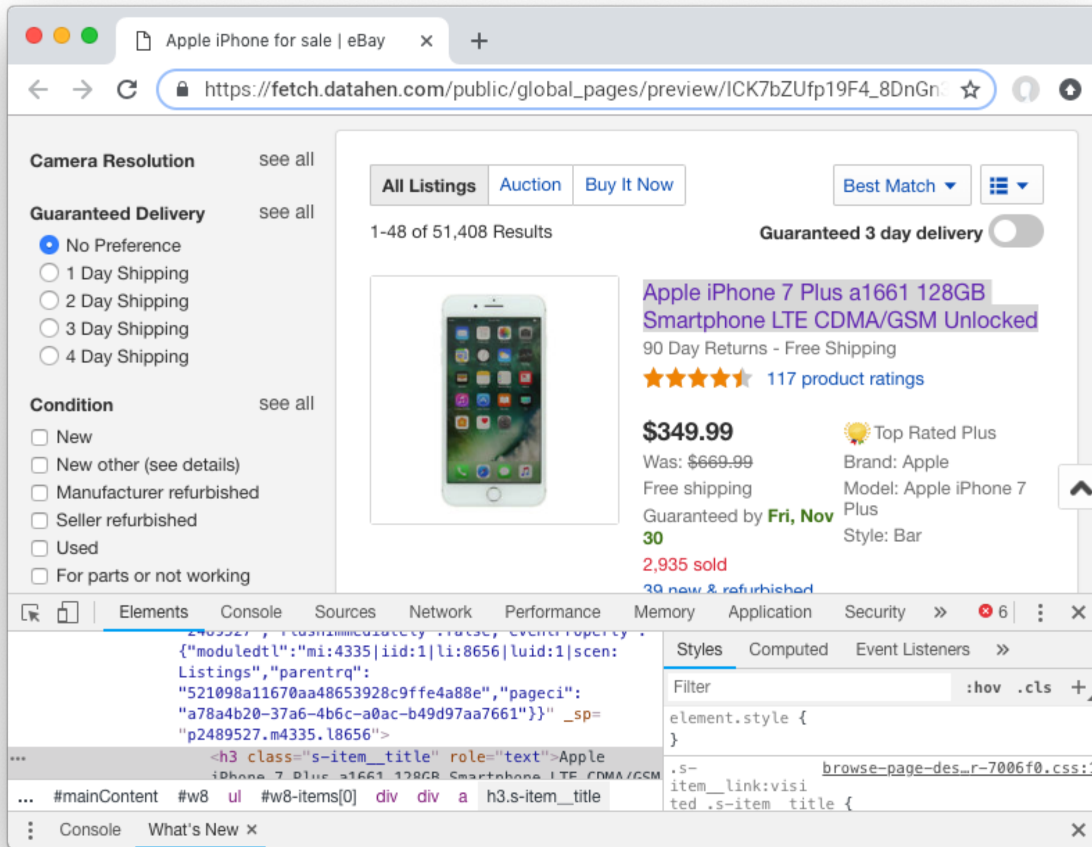
```
$ hen globalpage content www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e
Preview content url: "https://fetch.datahen.com/public/global_pages/preview/
↪ICK7bZUfp19F4_
↪8DnGn36tjGuS8Otfsg5QRDTXQyJTfQpGrc697HsAMiTlv6aKKH1x9rd8RYffluI7amPUvCAFOLDPlcH0Wmq3b0eiCJGUFi5xLhc
↪Owd5QCsUUS-
↪eW3OPMN5OPmYACwRJ0AJHESsjVTmsnfCw6EkMCUQlXmzz8Q2TMF8v1PHPv5185Z04w14rDY78E1UjdDOJO7W7jBK79JetWj5wUC
↪G7T9_9j0V5sA0dRn88wib19BIjboDHOczufyCv78x1f4njGMZZFwb"
```



Copy and paste the link that you got from the command line result, into a browser, and you will see the actual content of the the cache.

Previewing the cached page is a powerful thing when developing a parser script, because it allows you to get the actual page that the parser will execute on. This is much better than other kinds of web scraping frameworks out there, where you are developing scraper codes and "hoping" that the page scraped will be the same with what you are seeing.

Once you have the previewed content opened on your browser, let's look at it, and try to extract the listing names. You can use the Chrome Developer Tool (or other similar tools) to extract the DOM path.



What we are going to do next is to create a ruby script for parsing and extracting the listing names, prices, and the listing URLs, and enqueue additional pages, that we can parse on a later step.

Let's now create a ruby file, called listings.rb. For standard conventions, let's put this file in a directory called 'parsers.'

```
$ mkdir parsers
$ touch parsers/listings.rb
$ ls -alth
total 8
drwxr-xr-x  14 johndoe  staff   448B 27 Nov 20:01 .git
drwxr-xr-x   3 johndoe  staff    96B 27 Nov 20:01 parsers
drwxr-xr-x   6 johndoe  staff   192B 27 Nov 20:01 .
-rw-r--r--   1 johndoe  staff   123B 26 Nov 16:54 config.yaml
drwxr-xr-x   3 johndoe  staff    96B 26 Nov 16:15 seeder
```

(continues on next page)

(continued from previous page)

```
drwxr-xr-x  10 johndoe  staff   320B 26 Nov 15:59 ..
$ ls -alth parsers
total 0
drwxr-xr-x  3 johndoe  staff    96B 27 Nov 20:01 .
-rw-r--r--  1 johndoe  staff     0B 27 Nov 20:01 listings.rb
drwxr-xr-x  6 johndoe  staff   192B 27 Nov 20:01 ..
```

Next, type the following codes into the parsers/listings.rb file.

```
nokogiri = Nokogiri.HTML(content)
```

"content" is a reserved word that contains the content data. In this case, it is the same exact html content that you are currently also previewing from the globalpage. We support the use Nokogiri for parsing html/xml.

Let's add the codes that loops through the listing rows and extract each individual listing name, price, and listings URLs.

Get the group of listings from nokogiri.

```
listings = nokogiri.css('ul.b-list__items_nofooter li.s-item')
```

And then loop through the listing rows to extract the name, price, listing URL, and save the records to a DataHen collection.

```
listings.each do |listing|
    # initialize an empty hash
    product = {}

    # extract the information into the product hash
    product['title'] = listing.at_css('h3.s-item__title')&.text

    # extract the price
    product['price'] = listing.at_css('.s-item__price')&.text

    # extract the listing URL
    item_link = listing.at_css('a.s-item__link')
    product['url'] = item_link['href'] unless item_link.nil?

    # specify the collection where this record will be stored
    product['_collection'] = "listings"

    # save the product to the job's outputs
    outputs << product
end
```

"outputs" is a reserved word, that contains a list of records that will be stored onto the scrape job's output collection(s)

Once you've written the codes, your listings.rb script should look like this:

```
# initialize nokogiri
nokogiri = Nokogiri.HTML(content)

# get the group of listings
listings = nokogiri.css('ul.b-list__items_nofooter li.s-item')

# loop through the listings
listings.each do |listing|
```

(continues on next page)

(continued from previous page)

```
    # initialize an empty hash
    product = {}

    # extract the information into the product hash
    product['title'] = listing.at_css('h3.s-item__title')&.text

    # extract the price
    product['price'] = listing.at_css('.s-item__price')&.text

    # extract the listing URL
    item_link = listing.at_css('a.s-item__link')
    product['url'] = item_link['href'] unless item_link.nil?

    # specify the collection where this record will be stored
    product['_collection'] = "listings"

    # save the product to the job's outputs
    outputs << product
end
```

Now that you've created the listings.rb file, let's do a trial-run of this page to ensure that this page would execute properly on DataHen.

Let's use the parser try command on this script on the same GID.

```
$ hen parser try ebay parsers/listings.rb www.ebay.com-
↪4aa9b6bd1f2717409c22d58c4870471e
Trying parser script
getting Job Page
=========== Parsing Executed ===========
----------- Outputs: -----------
[
  {
    "title": "Apple iPhone 7 Plus a1661 128GB Smartphone LTE CDMA/GSM Unlocked",
    "price": "$349.99",
    "url": "https://www.ebay.com/itm/Apple-iPhone-7-Plus-a1661-128GB-Smartphone-LTE-
↪CDMA-GSM-Unlocked/201795245944?epid=232746597&hash=item2efbef1778:m:mks-K6wL_
↪LJV1VV0f3-E8ow:sc:USPSPriority!95113!US!-1&var=501834147259",
    "_collection": "listings"
  },
  {
    "title": "Apple iPhone XS MAX 256GB - All Colors - GSM & CDMA UNLOCKED",
    "price": "$1,199.00",
    "url": "https://www.ebay.com/itm/Apple-iPhone-XS-MAX-256GB-All-Colors-GSM-CDMA-
↪UNLOCKED/163267140481?epid=24023697465&hash=item26037adb81:m:m63pSpGuBZVZTiz-IS3A-
↪UA:sc:USPSPriorityMailPaddedFlatRateEnvelope!95113!US!-1&var=462501487846",
    "_collection": "listings"
  },
...
```

In the trial-run above, you saw that we were able to extract into several outputs correctly.

Seems like this is a successful trial run.

Let's now commit this code to Git.

```
$ git add .
$ git commit -m 'added listings parser'
```

(continues on next page)

(continued from previous page)

```
[master e1a8980] added listings parser
 1 file changed, 27 insertions(+)
 create mode 100644 parsers/listings.rb
```

Now that we're done with creating the parser script, there is only one thing left to do before this is ready to be run on DataHen. And that is to modify the config file so that it contains the references pointing to this file.

Add the parsers section of the config.yaml so that it looks like the following:

```
seeder:
 file: ./seeder/seeder.rb
 disabled: false # Optional. Set it to true if you want to disable
parsers:
 - page_type: listings
   file: ./parsers/listings.rb
   disabled: false # Optional
```

And let's commit this to Git, and push it to your remote Git repository.

```
$ git add .
$ git commit -m 'add listings parser to config'
[master 209dba3] add listings parser to config
 1 file changed, 6 insertions(+), 2 deletions(-)
$ git push origin master
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 1.12 KiB | 1.12 MiB/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To https://github.com/DataHenOfficial/ebay-scraper.git
   c32d451..209dba3  master -> master
```

Now that you've pushed the code to your remote Git repository, let's deploy the scraper again.

```
$ hen scraper deploy ebay
Deploying scraper. This may take a while...
{
 "id": 130,
 "scraper_id": 20,
 "commit_hash": "209dba31698b2146b9c841f0a91bfdd966f973aa",
 "git_repository": "https://github.com/DataHenOfficial/ebay-scraper.git",
 "git_branch": "master",
 "errors": null,
 "success": true,
 "created_at": "2018-11-28T03:05:02.220272Z",
 "config": {
  "parsers": [
   {
    "file": "./parsers/listings.rb",
    "page_type": "listings"
   }
  ],
  "seeder": {
   "file": "./seeder/seeder.rb"
  }
 }
}
```

Looks like a successful deploy.

Because you have a running scrape job (you've started this scrape job in Exercise 2), as soon as you deploy new code, the scrape job will immediately download and execute.

Let's now check for the stats on this job.

```
$ hen scraper stats ebay
{
 "job_id": 70,
 "pages": 1,
 "fetched_pages": 1,
 "to_fetch": 0,
 "fetching_failed": 0,
 "fetched_from_web": 1,
 "fetched_from_cache": 0,
 "parsed_pages": 1,          # Parsed pages is now 1
 "to_parse": 0,
 "parsing_failed": 0,
 "outputs": 48,              # We now have 48 output records
 "output_collections": 1,   # And we have 1 output collection
 "workers": 1,
 "time_stamp": "2018-11-28T03:06:04.959513Z"
}
```

If you look at the result of the command above, you see that our scraper has already parsed the listings page, and created 48 output records, and it also created one output collection.

Let's check the scraper log to see if there are any errors:

```
$ hen scraper log ebay
```

Seems like there are no errors so far.

Let's look further on what collections are created by the scraper.

```
$ hen scraper output collections ebay
[
 {
  "collection": "listings",
  "count": 48
 }
]
```

Just as we specified in the parser code, it saves the output records onto the listings collection.

Let's look at the output records inside the listings collection.

```
$ hen scraper output list ebay --collection listings
[
 {
  "_collection": "listings",
  "_created_at": "2018-11-28T03:05:56.510638Z",
  "_gid": "www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e",
  "_id": "016f6ca8010541539e13c687edd6cf91",
  "_job_id": 70,
  "price": "$239.99 to $339.99",
  "title": "Apple iPhone 7 32/128/256GB Factory Unlocked AT\u0026T Sprint Verizon T-
↪Mobile",
  "url": "https://www.ebay.com/itm/Apple-iPhone-7-32-128-256GB-Factory-Unlocked-AT-T-
↪Sprint-Verizon-T-Mobile/382231636268?
↪epid=232669182\u0026hash=item58fec7e92c:m:mGdfdcg2f2rqhQh7_
↪Aq4EYQ\u0026var=651061946000"
```

(continues on next page)

(continued from previous page)

```
},
{
 "_collection": "listings",
 "_created_at": "2018-11-28T03:05:56.510638Z",
 "_gid": "www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e",
 "_id": "029788f01c73460683f0910681348222",
 "_job_id": 70,
 "price": "$374.99",
 "title": "Apple iPhone 7 Plus a1661 256GB Smartphone LTE CDMA/GSM Unlocked",
 "url": "https://www.ebay.com/itm/Apple-iPhone-7-Plus-a1661-256GB-Smartphone-LTE-
↪CDMA-GSM-Unlocked/152410916266?epid=238409027\u0026hash=item237c6605aa:m:mks-K6wL_
↪LJV1VV0f3-E8ow:sc:USPSPriority!95113!US!-1\u0026var=451703657817"
},
...
```

We've got almost same exact output records with the ones on the trial-run. The difference is, now you can see "_gid", "_created_at", "_id", and "_job_id". These are metadatas that are automatically generated. Output metadata fields are usually starts with "_"(underscore) on their field names.

If you want to see one output record, you can run the following:

```
$ hen scraper output show ebay 97d7bcd021de4bdbb6818825703b26dd --collection listings
{
 "_collection": "listings",
 "_created_at": "2018-11-28T03:05:56.510638Z",
 "_gid": "www.ebay.com-4aa9b6bd1f2717409c22d58c4870471e",
 "_id": "97d7bcd021de4bdbb6818825703b26dd",
 "_job_id": 70,
 "price": "$259.99 to $344.99",
 "title": "Apple iPhone 7 - 32/128/256GB Unlocked GSM (AT\u0026T T-Mobile +More) 4G␣
↪Smartphone",
 "url": "https://www.ebay.com/itm/Apple-iPhone-7-32-128-256GB-Unlocked-GSM-AT-T-T-
↪Mobile-More-4G-Smartphone/142631642365?
↪epid=240455139\u0026hash=item21358224fd:m:miu7oWIlKlIyiHYnR76rFnA\u0026var=441608064608
↪"
}
```

We're nearing the end of the exercise. Let's cancel the current scrape job, as we will be creating a new job in the next exercise.

```
$ hen scraper job cancel ebay
{
 "id": 70,
 "scraper_name": "ebay",
 "scraper_id": 20,
 "created_at": "2018-11-26T22:06:54.399547Z",
 "freshness": null,
 "force_fetch": false,
 "status": "cancelled",
 "seeding_at": "2018-11-26T22:07:10.68643Z",
 "seeding_failed_at": null,
 "seeded_at": "2018-11-26T22:07:49.021679Z",
 "seeding_try_count": 1,
 "seeding_fail_count": 0,
 "seeding_error_count": 0,
 "worker_count": 1
}
```

Congratulations, you have completed exercise 3. You have learned to create a parser script that extracts a web page and save it to the DataHen output, and you have also learned how to use some useful commands related to scraper outputs and their collections.

The source codes that we've built throughout the exercise are located here https://github.com/DataHenOfficial/ebay-scraper/tree/exercise3.

In the next exercise, we'll be building upon this exercise to enqueue more pages to be scraped.

6.1.4 Exercise 4: Enqueue more pages, and pass variables to the next pages

In the last exercise, You have learned to create a parser script that extracts a web page and save it to the DataHen output, and you have also learned how to use some useful commands related to scraper outputs and their collections.

If you've skipped the last exercise, you can see the source code here: https://github.com/DataHenOfficial/ebay-scraper/tree/exercise4.

In this exercise, we will be building upon the previous exercise by enqueuing more pages from the listings parser, onto the details pages. We'll also be passing some page variables that we generated on the listings parser onto the the detail pages, so that the parser for the detail page can take advantage of it.

Let's add the following code inside the listings loop, just before the end of the loop in parsers/listings.rb:

```
# enqueue more pages to the scrape job
pages << {
    url: product['url'],
    page_type: 'details',
    vars: {
        title: product['title'],
        price: product['price']
    }
  }
```

Once done, the parsers/listings.rb should look like this:

```
# initialize nokogiri
nokogiri = Nokogiri.HTML(content)

# get the group of listings
listings = nokogiri.css('ul.b-list__items_nofooter li.s-item')

# loop through the listings
listings.each do |listing|
    # initialize an empty hash
    product = {}

    # extract the information into the product hash
    product['title'] = listing.at_css('h3.s-item__title')&.text

    # extract the price
    product['price'] = listing.at_css('.s-item__price')&.text

    # extract the listing URL
    item_link = listing.at_css('a.s-item__link')
    product['url'] = item_link['href'] unless item_link.nil?

    # specify the collection where this record will be stored
    product['_collection'] = "listings"
```

(continues on next page)

(continued from previous page)

```
    # save the product to the outputs.
    outputs << product

    # enqueue more pages to the scrape job
    pages << {
        url: product['url'],
        page_type: 'details',
        vars: {  # adding vars to this page
            title: product['title'],
            price: product['price']
        }
      }
end
```

Let's now trial-run this page on a GID of the page that we've used on the past exercise.

```
$ hen parser try ebay parsers/listings.rb www.ebay.com-
↪4aa9b6bd1f2717409c22d58c4870471e
Trying parser script
getting Job Page
=========== Parsing Executed ===========
----------- Outputs: -----------
[
  {
    "title": "Apple iPhone 7 Plus a1661 128GB Smartphone LTE CDMA/GSM Unlocked",
    "price": "$349.99",
    "url": "https://www.ebay.com/itm/Apple-iPhone-7-Plus-a1661-128GB-Smartphone-LTE-
↪CDMA-GSM-Unlocked/201795245944?epid=232746597&hash=item2efbef1778:m:mks-K6wL_
↪LJV1VV0f3-E8ow:sc:USPSPriority!95113!US!-1&var=501834147259",
    "_collection": "listings"
  },
...
----------- New Pages to Enqueue: -----------
[
  {
    "url": "https://www.ebay.com/itm/Apple-iPhone-7-Plus-a1661-128GB-Smartphone-LTE-
↪CDMA-GSM-Unlocked/201795245944?epid=232746597&hash=item2efbef1778:m:mks-K6wL_
↪LJV1VV0f3-E8ow:sc:USPSPriority!95113!US!-1&var=501834147259",
    "page_type": "details",
    "vars": {
      "title": "Apple iPhone 7 Plus a1661 128GB Smartphone LTE CDMA/GSM Unlocked",
      "price": "$349.99"
    }
  },
  {
    "url": "https://www.ebay.com/itm/Apple-iPhone-XS-MAX-256GB-All-Colors-GSM-CDMA-
↪UNLOCKED/163267140481?epid=24023697465&hash=item26037adb81:m:m63pSpGuBZVZTiz-IS3A-
↪UA:sc:USPSPriorityMailPaddedFlatRateEnvelope!95113!US!-1&var=462501487846",
    "page_type": "details",
    "vars": {
      "title": "Apple iPhone XS MAX 256GB - All Colors - GSM & CDMA UNLOCKED",
      "price": "$1,199.00"
    }
  },
...
```

The trial-run seems to be successful, and you now see a new pages to enqueue, as well as the page vars inside each of them.

Let's now commit this code and push it to the remote Git repository.

```
$ git add .
$ git commit -m 'modified listings parser to enqueue more pages'
[master 332e06d] modified listings parser to enqueue more pages
 1 file changed, 10 insertions(+)
$ git push origin master
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 567 bytes | 567.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/DataHenOfficial/ebay-scraper.git
   209dba3..332e06d  master -> master
```

Now deploy it, and start a new scrape job, so that we have some page contents in the shared cache.

```
$ hen scraper deploy ebay
Deploying scraper. This may take a while...
{
 "id": 131,
 "scraper_id": 20,
 "commit_hash": "332e06d01f073ac00840184cd06c826429b3b55c",
...

$ hen scraper start ebay
Starting a scrape job...
{
 "id": 88,
 "scraper_id": 20,
...
```

Let's check the scraper stats now:

```
$ hen scraper stats ebay
{
 "job_id": 88,
 "pages": 49,
 "fetched_pages": 3,
 "to_fetch": 46,
 "fetching_failed": 0,
 ...
}
```

Looks like we now have 49 pages in this job. This means we have successfully modified the parsers/listings.rb to enqueue more pages.

Let's see what pages do we have in this scrape job:

```
$ hen scraper page list ebay
[
 {
  "gid": "www.ebay.com-eb1b04c3304ff741b5dbd3234c9da75e",
  "job_id": 88,
```

(continues on next page)

(continued from previous page)

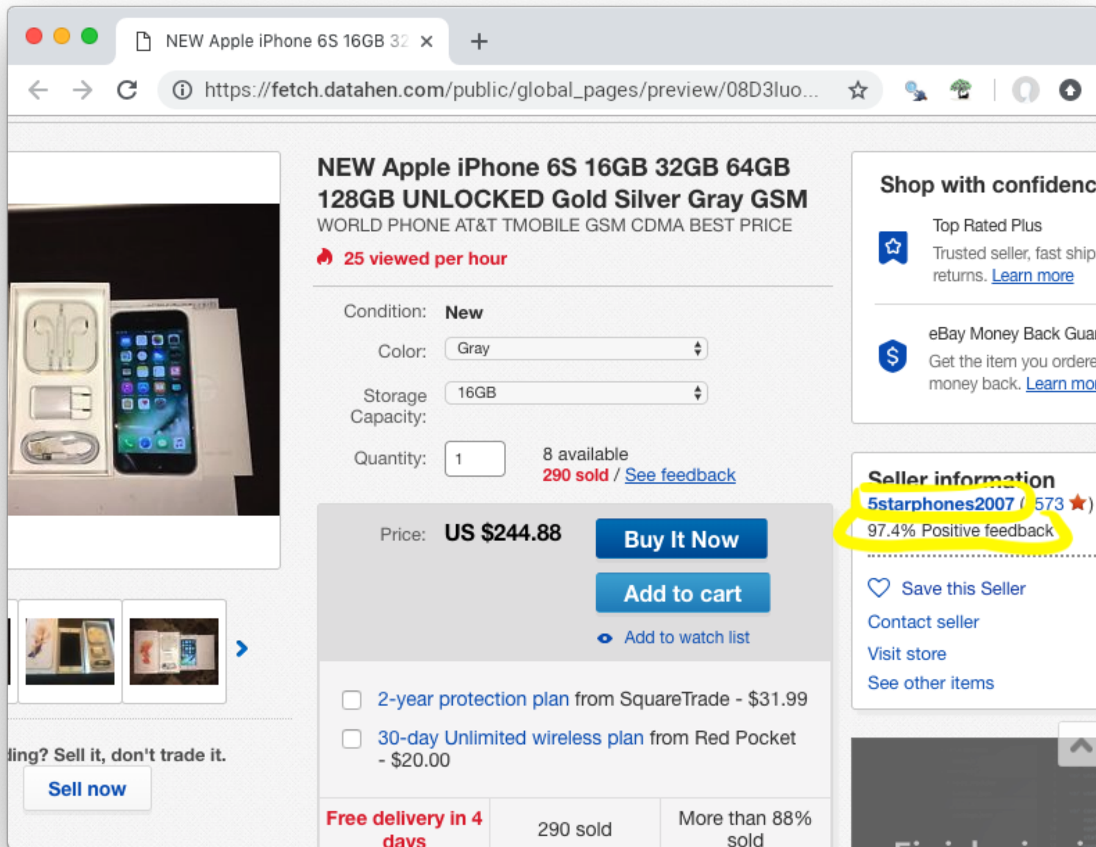
```
  "page_type": "details",
  "method": "GET",
  "url": "https://www.ebay.com/itm/NEW-Apple-iPhone-6S-16GB-32GB-64GB-128GB-UNLOCKED-
↪Gold-Silver-Gray-GSM/153152283464?epid=240420050\u0026hash=item23a8966348%3Am
↪%3AmN52WSYRFZEr3HNCPAgfZfQ\u0026var=453004437286",
  ...
  "vars": {
   "price": "$275.99",
   "title": "Apple iPhone 7 128GB 4.7\" Retina Display 4G GSM BLACK UNLOCKED␣
↪Smartphone SRF"
  },
...
```

Looks like it correctly enqueued the page and the vars.

Now, let's preview the content of the GID from the above result.

```
$ hen globalpage content www.ebay.com-eb1b04c3304ff741b5dbd3234c9da75e
Preview content url: "https://fetch.datahen.com/public/global_pages/preview/
↪V58Ts9JDocg7qyjT5cxkIL5Dk_
↪UEuEyOQXaHbxBdNJiHmsP2nhE17a8gzwXO8RYUNJ2X1K7PWh0ziIXO7ORylLVLppiufL1edZChFFr8quZuivTJql7B3z_
↪btQA9p1wQqiowc4U90VZtP4bOrShW9v9fA6V4vedfpJL9TM5G8QR-
↪zYhUbrwAIpB4WDLALpPz19ZPZ1rjt8OVYJ2GRDCgr2fPb1S2uBKMI35k1oXO3Cr0yf8St_
↪M5lZqUbEYyqVLPCwDC7ZFk1J_
↪KxQvzgD6vWqmKRoGk1XVS7gxPBusNip47gANzPbo9p3wjWHjqaQhhPb0UFXKBithJ"
```

Now, let's create a details parser and extract the seller's username and their feedback.



Create the parsers/details.rb file and Insert the following code:

```
# initialize nokogiri
nokogiri = Nokogiri.HTML(content)

# get the seller username
seller = nokogiri.at_css('.si-inner .mbg-nw')&.text

# get the seller's feedback
feedback = nokogiri.at_css('.si-inner #si-fb')&.text

# save it into outputs
outputs << {
    _collection: 'products',
    title: page['vars']['title'],
    price: page['vars']['price'],
    seller: seller,
    feedback: feedback
}
```

Let's now give this parser a try on one of the details page:

```
$ hen parser try ebay parsers/details.rb www.ebay.com-eb1b04c3304ff741b5dbd3234c9da75e
```

(continues on next page)

(continued from previous page)

```
Trying parser script
getting Job Page
=========== Parsing Executed ===========
----------- Outputs: -----------
[
  {
    "_collection": "products",
    "title": "Apple iPhone 8 Plus a1897 64GB Smartphone GSM Unlocked",
    "price": "$550.99 to $599.99",
    "seller": "mywit",
    "feedback": "97.6% Positive feedback"
  }
]
```

This trial-run looks really good. That we're able to extract the details page without error, and assumed that we would be able to save the output to the products collection.

Let's now commit this details parser.

```
$ git add .
$ git commit -m 'added details parser'
[master 59bab08] added details parser
 1 file changed, 17 insertions(+)
 create mode 100644 parsers/details.rb
```

Let's not forget to update the config yaml file so that it points to this parser file. The config file should look like this now:

```
seeder:
  file: ./seeder/seeder.rb
  disabled: false
parsers:
  - page_type: listings
    file: ./parsers/listings.rb
    disabled: false
  - page_type: details
    file: ./parsers/details.rb
    disabled: false
```

Let's commit the file, and push it to the remote Git repository.

```
$ git add .
$ git commit -m 'added details parser config'
[master 97dac60] added details parser config
 1 file changed, 4 insertions(+), 1 deletion(-)
$ git push origin master
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 1008 bytes | 1008.00 KiB/s, done.
Total 7 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/DataHenOfficial/ebay-scraper.git
   332e06d..97dac60  master -> master
```

Next, let's deploy it to DataHen. Since we already have the current job running, we don't need to do anything else, and the scraper will immediately execute the newly deployed code.

```
$ hen scraper deploy ebay
Deploying scraper. This may take a while...
{
 "id": 132,
 "scraper_id": 20,
 "commit_hash": "97dac606b24a8c818f3234419cb9180999bc9d71",
 "git_repository": "https://github.com/DataHenOfficial/ebay-scraper.git",
 "git_branch": "master",
 "errors": null,
 "success": true,
 "created_at": "2018-11-28T06:05:48.866609Z",
 "config": {
  "parsers": [
   {
    "file": "./parsers/listings.rb",
    "page_type": "listings"
   },
   {
    "file": "./parsers/details.rb",
    "page_type": "details"
   }
  ],
  "seeder": {
   "file": "./seeder/seeder.rb"
  }
 }
}
```

Let's check to see the scraper stats:

```
$ hen scraper stats ebay
{
 ...
 "parsed_pages": 49,
 "to_parse": 0,
 "parsing_failed": 0,
 "outputs": 96,
 "output_collections": 2,
 ...
}
```

This looks good, as it shows that there are 2 collections, and a total of 96 output records.

Let's also check the scraper log to see if there are any errors:

```
$ hen scraper log ebay
```

Seems like there are no errors so far.

Let's look to see if the products collection has been correctly created:

```
$ hen scraper output collection ebay
[
 {
  "collection": "listings",
  "count": 48
 },
 {
```

(continues on next page)

(continued from previous page)

```
  "collection": "products",
  "count": 48
 }
]
```

Looks good. Now, let's look inside the products collection and see if the outputs are correctly saved:

```
$ hen scraper output list ebay --collection products
[
 {
  "_collection": "products",
  "_created_at": "2018-11-28T06:15:13.061168Z",
  "_gid": "www.ebay.com-82d133d0dba1255c7424f99ab79776e9",
  "_id": "ac3c99ab8d3846a98d3317660259db06",
  "_job_id": 91,
  "feedback": "99.8% Positive feedback",
  "price": "$269.98",
  "seller": "overdrive_brands",
  "title": "Apple iPhone 7 32GB Smartphone AT\u0026T T-Mobile Unlocked 4G LTE Black␣
↪Red Rose Gold"
 },
 {
  "_collection": "products",
  "_created_at": "2018-11-28T06:15:13.47726Z",
  "_gid": "www.ebay.com-0cc7480bd6afecb10cf81bdc5904ea74",
  "_id": "7be4ee3e544e4e5db9115215572259b9",
  "_job_id": 91,
  "feedback": "99.9% Positive feedback",
  "price": "$569.00",
  "seller": "alldayzip",
  "title": "Apple iPhone 8 64GB RED \u0026 All Colors! GSM \u0026 CDMA UNLOCKED!!␣
↪BRAND NEW! Warranty!"
 },
...
```

The output looks wonderful.

Congratulations, you have completed exercise 4. You have learned to enqueue more pages from the listings parser onto detail pages. You have also learned about passing some page variables that were generated on the listings parser onto the the detail pages. The details parsers then combined the data from the page vars and the extracted data from the details page, into the "products" output collection.

The source codes that we've built throughout the exercise are located here https://github.com/DataHenOfficial/ebay-scraper/tree/exercise4.

In the next exercise, you will be learning how to export output by creating Exporters.

6.1.5 Exercise 5: Exporting outputs

In the last exercise, You have learned to do some more advanced techniques of creating parsers, and also saved the records into the output collection.

If you've skipped the last exercise, you can see the source code here: https://github.com/DataHenOfficial/ebay-scraper/tree/exercise4.

In this exercise, we will be exporting outputs by creating Exporters. In DataHen, an Exporter is a set of configurations that allows you to Export something. We have several different kinds of Exporters. Please read the documentation for

further details

We're going to export the *products* collection (which you've created in Exercise 4) into a JSON file that we can download, once the export process has finished.

Before exporting all the outputs of the *products* collection, it is a good idea to sample a smaller number of records first, to make sure the outputs are looking good.

To do that, let's create an exporter configuration file, called *products_last10_json.yaml*. For standard conventions, let's put this file in a directory called 'exporters'.

First you need to create the *exporters* directory on the project root directory.

```
$ mkdir exporters
$ touch exporters/products_last10_json.yaml
$ ls -alth exporters
total 0
drwxr-xr-x  3 johndoe  staff    96B 13 Jan 00:56 .
-rw-r--r--  1 johndoe  staff     0B 13 Jan 00:56 products_last10_json.yaml
drwxr-xr-x  7 johndoe  staff   224B 13 Jan 00:55 ..
```

Next let's put the following content into the products_last10_json.yaml file:

```
exporter_name: products_last10_json # Must be unique
exporter_type: json
collection: products
write_mode: pretty_array # can be `line`,`pretty`, `pretty_array`, or `array`
limit: 10 # limits to how many records to export
offset: 0 # offset to where the exported record will start from
```

The above exporter means, we want to export only the last 10 outputs from the *products* collection

Next, let's modify the config.yaml file, so that it knows the location of the exporter file.

Your config.yaml should now look like the following:

```
seeder:
 file: ./seeder/seeder.rb
 disabled: false
parsers:
 - page_type: listings
   file: ./parsers/listings.rb
   disabled: false
 - page_type: details
   file: ./parsers/details.rb
   disabled: false
# add the following lines below...
exporters:
 - file: ./exporters/products_last10_json.yaml
   disabled: false
```

Now, let's commit these files and push this to the remote repository.

```
$ git add .
$ git commit -m 'added products_last10_json.yaml exporter'
[master ab7ab52] added products_last10_json.yaml exporter
 2 files changed, 10 insertions(+)
 create mode 100644 exporters/products_last10_json.yaml
$ git push origin master
Counting objects: 5, done.
```

(continues on next page)

(continued from previous page)

```
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 742 bytes | 742.00 KiB/s, done.
Total 5 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/DataHenOfficial/ebay-scraper.git
   7bd6091..ab7ab52  master -> master
```

Let's now deploy the scraper.

```
$ hen scraper deploy ebay
Deploying scraper. This may take a while...
{
 "id": 737,
 "scraper_id": 20,
 ...
 "config": {
  "exporters": [
   {
    "collection": "products",
    "exporter_name": "products_last10_json",
    "exporter_type": "json",
    "limit": 10,
    "offset": 0,
    "write_mode": "pretty_array"
   }
  ],
 ...
}
```

You've now deployed the newest code.

To make sure that the exporters are deployed correctly, you can check a list of available exporters on this scraper:

```
$ hen scraper exporter list ebay
[
 {
  "collection": "products",
  "exporter_name": "products_last10_json",
  "exporter_type": "json",
  "limit": 10,
  "offset": 0,
  "write_mode": "pretty_array"
 }
]
```

Next, let's start the exporter:

```
$ hen scraper exporter start ebay products_last10_json
{
 "id": "c700cb749f4e45eeb53609927e21da56", # Export ID here
 "job_id": 852,
 "scraper_id": 20,
 "exporter_name": "products_last10_json",
 "exporter_type": "json",
 "config": {
  "collection": "products",
```

(continues on next page)

(continued from previous page)

```
  "exporter_name": "products_last10_json",
  "exporter_type": "json",
  "limit": 10,
  "offset": 0,
  "write_mode": "pretty_array"
 },
 "status": "enqueued", # the status of the export
 "created_at": "2019-01-13T06:19:56.815979Z"
}
```

When you start an exporter, it creates an export record like the above.

You can check the status of the export by specifying the export ID, in this case: **c700cb749f4e45eeb53609927e21da56**

Let's now check the status of the export:

```
$ hen scraper export show c700cb749f4e45eeb53609927e21da56
{
 "id": "c700cb749f4e45eeb53609927e21da56",
 ...
 "status": "done", # the export is done
 "created_at": "2019-01-13T06:19:56.815979Z",
 "progress": 100, # Progress is at 100%, again, it means it is done.
 "exported_records": 10,
 "to_export": 10,
 "exporting_at": "2019-01-13T06:19:59.794454Z",
 "exported_at": "2019-01-13T06:20:01.125188Z",
 "file_name": "c700cb749f4e45eeb53609927e21da56.tgz"
}
```

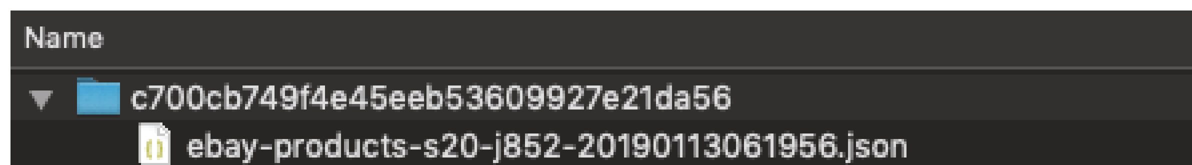
As you can see above by looking at the status, it is done. It also shows the export's file_name: **c700cb749f4e45eeb53609927e21da56.tgz**

Let's now download the export, by specifying the export ID, like so:

```
$ hen scraper export download c700cb749f4e45eeb53609927e21da56
Download url: "https://f002.backblazeb2.com/file/exports-fetch-datahen/
↪c700cb749f4e45eeb53609927e21da56.tgz?Authorization=blablabla"
```

You're now seeing the download URL, which allows you to copy and paste into your browser so that you'll download the actual file.

If you uncompress this .tgz file, you'll see the following directory with one content:



And when you open the json file, you'll see the following content with the 10 most recent records from the products collection:

```
[{
  "_collection": "products",
  "_created_at": "2019-01-13T01:17:21.925076Z",
  "_gid": "www.ebay.com-384cfde09349adaa48f78c44b5f840db",
  "_id": "574d346fb77c4d669896f15f6b40f169",
```

(continues on next page)

(continued from previous page)

```
  "_job_id": 852,
  "feedback": "97.6% Positive feedback",
  "price": "$183.99 to $235.99",
  "seller": "x-channel",
  "title": "New *UNOPENDED* Apple iPhone SE - 16/64GB 4.0\" Unlocked Smartphone"
},
{
  "_collection": "products",
  "_created_at": "2019-01-13T01:16:53.494641Z",
  "_gid": "www.ebay.com-e5681a88f980aa906f7c4414e9120685",
  "_id": "231034ec579c4b329b0eaf8ebc642ed1",
  "_job_id": 852,
  "feedback": "98.3% Positive feedback",
  "price": "$299.99",
  "seller": "bidallies",
  "title": "Apple iPhone 7 Plus 32GB Factory Unlocked 4G LTE iOS WiFi Smartphone"
},
]
```

Now that this file looks good, let's create another exporter to export the all of records.

This is a very simple process, you just need to create another exporter file called *products_json.yaml* with the following content:

```
exporter_name: products_json
exporter_type: json
collection: products
```

Notice how we're not putting anything such as *limit*, *offset*, and even *write_mode*. This is because the default value for *write_mode* is *pretty_array*.

Don't forget to update the config.yaml file to look like the following:

```
seeder:
 file: ./seeder/seeder.rb
 disabled: false
parsers:
 - page_type: listings
   file: ./parsers/listings.rb
   disabled: false
 - page_type: details
   file: ./parsers/details.rb
   disabled: false
exporters:
 - file: ./exporters/products_last10_json.yaml
   disabled: false
# Add the following lines below...
 - file: ./exporters/products_json.yaml
   disabled: false
```

Let's now commit and push this to the remote repos:

```
$ git add .
$ git commit -m 'added full products exporter'
[master 2f453eb] added full products exporter
 2 files changed, 5 insertions(+)
 create mode 100644 exporters/products_json.yaml
```

(continues on next page)

(continued from previous page)

```
$ git push origin master
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 569 bytes | 569.00 KiB/s, done.
Total 5 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/DataHenOfficial/ebay-scraper.git
   ab7ab52..2f453eb  master -> master
```

Let's now deploy it:

```
$ hen scraper deploy ebay
Deploying scraper. This may take a while...
{
 ...
 "config": {
  "exporters": [
   {
    "collection": "products",
    "exporter_name": "products_last10_json",
    "exporter_type": "json",
    "limit": 10,
    "offset": 0,
    "write_mode": "pretty_array"
   },
   {
    "collection": "products",
    "exporter_name": "products_json",
    "exporter_type": "json",
    "limit": null,
    "offset": null
   }
  ],
  ...
}
```

Once deployed, let's see what exporters are available to be used on the scraper:

```
$ hen scraper exporter list ebay
[
 {
  "collection": "products",
  "exporter_name": "products_last10_json",
  "exporter_type": "json",
  "limit": 10,
  "offset": 0,
  "write_mode": "pretty_array"
 },
 {
  "collection": "products",
  "exporter_name": "products_json",
  "exporter_type": "json",
  "limit": null,
  "offset": null
 }
]
```

Great, we've verified that we have indeed two exporters.

Now, start the exporter that we've just recently created:

```
$ hen scraper exporter start ebay products_json
{
 "id": "6bfd70f5c9f346b3a623cab50ea8a84c",
 "job_id": 852,
 "scraper_id": 20,
 "exporter_name": "products_json",
 "exporter_type": "json",
 "config": {
  "collection": "products",
  "exporter_name": "products_json",
  "exporter_type": "json",
  "limit": null,
  "offset": null
 },
 "status": "enqueued",
 "created_at": "2019-01-13T06:53:30.698121Z"
}
```

Next, let's check the status of the export, by specifying the export ID:

```
$ hen scraper export show 6bfd70f5c9f346b3a623cab50ea8a84c
{
 "id": "6bfd70f5c9f346b3a623cab50ea8a84c",
 "job_id": 852,
 "scraper_id": 20,
 "scraper_name": "ebay",
 "exporter_name": "products_json",
 "exporter_type": "json",
 "config": {
  "collection": "products",
  "exporter_name": "products_json",
  "exporter_type": "json",
  "limit": null,
  "offset": null
 },
 "status": "done",
 "created_at": "2019-01-13T06:53:30.698121Z",
 "progress": 100,
 "exported_records": 48,
 "to_export": 48,
 "exporting_at": "2019-01-13T06:53:33.501493Z",
 "exported_at": "2019-01-13T06:53:35.141981Z",
 "file_name": "6bfd70f5c9f346b3a623cab50ea8a84c.tgz"
}
```

As you can see, the status is now done. Next, let's download the export:

```
$ hen scraper export download 6bfd70f5c9f346b3a623cab50ea8a84c
Download url: "https://f002.backblazeb2.com/file/exports-fetch-datahen/
↪6bfd70f5c9f346b3a623cab50ea8a84c.tgz?Authorization=blablabla"
```

Now, if you open and count the records inside the file, you will notice that it will have 48 records. Which is the exact same count that is shown in the *products* collection below:

```
$ hen scraper output collection ebay
[
 {
  "collection": "listings",
  "count": 48
 },
 {
  "collection": "products",
  "count": 48 # the exported count is the same as this
 }
]
```

You have now created and used two exporters successfully.

Let's try to use a different kind of exporter. This time, let's use the CSV Exporter. Create another file, called *products_csv.yaml* with the following content:

```
exporter_name: products_csv
exporter_type: csv
collection: products
fields:
 - header: "gid"
   path: "_gid"
 - header: "created_at"
   path: "_created_at"
 - header: "title"
   path: "title"
 - header: "price"
   path: "price"
 - header: "feedback"
   path: "feedback"
 - header: "seller"
   path: "seller"
```

Since you've done this a couple of times now, go ahead and commit and deploy this exporter by yourself. Don't forget to add the path to this file, into your config.yaml file.

We're nearing the end of the exercise, but before that, I want to show you two more things.

First, DataHen allows you to automatically run the exporter when the scrape job is done. To do this, simply add `start_on_job_done: true` to your exporter configuration. Let's update the file *products_csv.yaml* so that it looks like following content:

```
exporter_name: products_csv
exporter_type: csv
collection: products
start_on_job_done: true # we added this field
fields:
 - header: "gid"
   path: "_gid"
 - header: "created_at"
   path: "_created_at"
 - header: "title"
   path: "title"
 - header: "price"
   path: "price"
 - header: "feedback"
   path: "feedback"
```

(continues on next page)

(continued from previous page)

```
- header: "seller"
  path: "seller"
```

Now that you've done the above, whenever your scrape job is done, the products_csv exporter will automatically be run.

Last thing to show before you go. Let's look at a list of all the exports that were exported from all scrapers in your account:

```
$ hen scraper export list
[
 {
  "id": "fe70e697354643429712c9880fb3678e",
  ...
 },
 {
  "id": "c700cb749f4e45eeb53609927e21da56",
  ...
 },
...
```

You can even filter all the exports based on the scraper name:

```
$ hen scraper export list --scraper-name ebay
[
 {
  "id": "6bfd70f5c9f346b3a623cab50ea8a84c",
  ...
 },
...
```

Congratulations, you have completed exercise 5.

You have learned how to create an exporter that can export partial records, as well as an exporter that exports the full records.

DataHen offers several kinds of exporters including CSV that exports to the CSV format, and Content exporter that allows you to export contents, such as files, images, pdf, etc. For more information please visit the documentation

The source codes that we've built throughout the exercise are located here https://github.com/DataHenOfficial/ebay-scraper/tree/exercise5.

6.1.6 Exercise 6: Create a finisher script

In the last exercise we went through creating an exporter to export scraper output. In this exercise we are going to take a look at creating a finisher script. A finisher script is a script that runs after a scraper job is done. With a finisher script you can do things like create summaries or run QA on your scraped data. Both of which we will show you how to do. Let's first create a finisher script to create a summary that shows the total number of listings. Create a folder called finisher in your project root directory and then create a file called finisher.rb inside with the following:

```
collections = DataHen::Client::ScraperJobOutput.new.collections("ebay")
collection = collections.find{|collection| collection['collection'] == "listings" }
if collection
  total = collection["outputs"]
   outputs << {
     "_collection" => "summary",
```

(continues on next page)

(continued from previous page)

```
    "total_listings" => total
  }
else
  puts "no listings collection found"
end
```

Basically we are using the DataHen gem to find all the collections for our ebay scraper and selecting the, "listings" collection. We then get the total number of listings inside this collection and save it to a new collection called, "summary." Next, let's add our finisher to the config.yaml file so that it looks like the following:

```
seeder:
 file: ./seeder/seeder.rb
 disabled: false
parsers:
 - page_type: listings
   file: ./parsers/listings.rb
   disabled: false
 - page_type: details
   file: ./parsers/details.rb
   disabled: false
exporters:
 - file: ./exporters/products_last10_json.yaml
   disabled: false
 - file: ./exporters/products_json.yaml
   disabled: false
finisher:
  file: ./finisher/finisher.rb
  disabled: false
```

Now that we have updated our config, let's give this finisher a try by running the following:

```
hen finisher try ebay finisher/finisher.rb

Trying seeder script
=========== Seeding Executed ===========
----------------------------------------
Would have saved 1 out of 1 Outputs
[
  {
    "_collection": "summary",
    "total_listings": 39
  }
]
```

Looks like this worked! The finisher would have saved a summary collection with the total number of listings. Let's now commit and push this to the remote repos:

```
$ git add .
$ git commit -m 'added finisher script'
[master 2f453eb] added finisher script
 1 files changed, 5 insertions(+)
 create mode 100644 finisher/finisher.rb
$ git push origin master
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
```

(continues on next page)

(continued from previous page)

```
Writing objects: 100% (5/5), 569 bytes | 569.00 KiB/s, done.
Total 5 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/DataHenOfficial/ebay-scraper.git
   ab7ab52..2f453eb  master -> master
```

Let's now deploy it:

```
$ hen scraper deploy ebay
Deploying scraper. This may take a while...
{
 ...
 "config": {
   "finisher": {
   "file": "./finisher/finisher.rb"
  },
  ...
}
```

And then start the scraper.

```
$ hen scraper start ebay
Starting a scrape job...
```

Give the scraper a few minutes and then check the status. We are looking for the "finisher_status" to be "done." It should look something like the following:

```
hen scraper stats ebay
{
 "scraper_name": "ebay",
 "job_id": 10066,
 "job_status": "done",
 "seeding_status": "done",
 "finisher_status": "done",
 "pages": 40,
 "to_fetch": 0,
 "fetching": 0,
 "fetching_failed": 0,
 "fetching_dequeue_failed": 0,
 "to_parse": 0,
 "parsing_started": 0,
 "parsing": 0,
 "parsed": 40,
 "parsing_failed": 0,
 "parsing_dequeue_failed": 0,
 "limbo": 0,
 "fetched": 40,
 "fetched_from_web": 0,
 "fetched_from_cache": 40,
 "outputs": 79,
 "output_collections": 0,
 "standard_workers": 1,
 "browser_workers": 0,
 "time_stamp": "2019-08-15T17:32:48.771103Z"
}
```

Once the finisher has run we can check our collections with the following:

```
hen scraper output collections ebay
[
 {
  "job_id": 10066,
  "collection": "listings",
  "outputs": 39
 },
 {
  "job_id": 10066,
  "collection": "products",
  "outputs": 39
 },
 {
  "job_id": 10066,
  "collection": "summary",
  "outputs": 1
 }
]
```

Looks like our finisher script worked! We now have a summary collection which shows the number of listings.

Next let's take a look at adding some QA to our finisher so we can validate the scraper results. We will use the 'dh_easy-qa' Gem which is a Ruby Gem that allows for doing QA on DataHen script outputs. First, create a file called Gemfile in the project root directory with the following:

```
gem 'dh_easy-qa'
```

After creating this file, run the following command in the project root directory:

```
bundle
```

This will install the 'dh_easy-qa' Gem. You should see something like the following output.

```
Resolving dependencies...
Using dh_easy-qa 0.0.26
Using bundler 1.17.3
Bundle complete! 1 Gemfile dependency, 2 gems now installed.
```

Now we need to create a file called dh_easy.yaml, also in the project root directory, with the following:

```
qa:
  individual_validations:
    url:
      required: true
      type: Url
    title:
      required: true
      type: String
```

This dh_easy.yaml file is where we can define validations for the 'dh_easy-qa' Gem to use. In this example, we are going to do validation on the listings collection output. Specifically, we are validating that the "url" field is present and is of type "Url" and that the "title" field is present and is a String.

Next, we just need to add some code to make the validator run in our finisher script. Update the finisher.rb file so it looks like the following:

```
require "dh_easy/qa'

collections = DataHen::Client::ScraperJobOutput.new.collections("ebay")
collection = collections.find{|collection| collection['collection'] == "listings" }
if collection
  total = collection["outputs"]
   outputs << {
     "_collection" => "summary",
     "total_listings" => total
   }
else
   puts "no listings collection found"
end

vars = { "scraper_name" => "ebay", "collections" => ["listings"]}
DhEasy::Qa::Validator.new.validate_internal(vars, outputs)
```

We are adding a line that loads the "dh_easy-qa" Gem using require and then doing validation on the listings collection of our ebay scraper. Let's try our finisher again. Run the following:

```
hen finisher try ebay finisher/finisher.rb

Trying finisher script
1
2
validating scraper: ebay
Validating collection: listings
data count 39
=========== Finisher Executed ===========
----------------------------------------
Would have saved 2 out of 2 Outputs
[
  {
    "_collection": "summary",
    "total_listings": 39
  },
  {
    "pass": "true",
    "_collection": "ebay_listings_summary",
    "total_items": 39
  }
]
```

Ok, great! The "pass": "true" means that the validations all passed. Feel free to edit validation rules in the dh_easy.yaml file. There are more details and rules in the, "How to write a QA script to ingest and parse outputs from multiple scrapers" tutorial in the, "Advanced Tutorials" section.

Let's now commit this update, push it to master, deploy it, and start the scraper again by running the following commands:

```
$ git add .
$ git commit -m 'added qa to listings in finisher script'
$ git push origin master
$ hen scraper deploy ebay
$ hen scraper start ebay
```

Give the scraper a few minutes to finish. You can check the status with the following:

```
hen scraper stats ebay
```

Once the “finisher_status” if “done,” we can check the collections again for the QA summary with the following command:

```
hen scraper output collections ebay
[
 {
  "job_id": 10066,
  "collection": "listings",
  "outputs": 39
 },
 {
  "job_id": 10066,
  "collection": "products",
  "outputs": 39
 },
 {
  "job_id": 10066,
  "collection": "summary",
  "outputs": 1
 },
 {
  "pass": "true",
  "collection": "ebay_listings_summary",
  "total_items": 39
 }
]
```

Great, it looks like we now have our summary as well as our QA summary present, which means the finisher script has run successfully.

CHAPTER 7

Advanced Tutorials

7.1 How to write a QA script to ingest and parse outputs from multiple scrapers

7.1.1 Step 1: Setup

In this tutorial we are going to go over the steps to do QA (Quality Assurance) on the output from your scrapers so you can validate that the data you scrape is correct. We do this by creating another "QA" scraper that will ingest the output from your initial scraper and perform QA on it using rules that you define.

For this example, we are going to be creating a QA scraper for the, "Simple Ebay scraper" that we previously created in, "Coding Tutorials." Let's create an empty directory and name it 'ebay-scraper-qa'

```
$ mkdir ebay-scraper-qa
```

Next let's go into the directory and initialize it as a Git repository

```
$ cd ebay-scraper-qa
$ git init .
Initialized empty Git repository in /Users/workspace/ebay-scraper-qa/.git/
```

Create a file called Gemfile inside the ebay-scraper-qa directory with the following line. This will add the dh_easy-qa Gem, which is the Gem that handles the QA.

```
gem 'dh_easy-qa'
```

Now run the following command in the ebay-scraper-qa directory to install this gem:

```
bundle
```

Next let's make a yaml file that will contain the configuration for the QA. Create a file called dh_easy.yaml inside the ebay-scraper-qa directory with the following contents:

```
qa:
  scrapers:
    ebay-scraper:
      - listings
```

This is the configuration we use to tell the QA what scrapers to use and which collections within each scraper to perform QA on. In this example, we are going to be doing QA on our ebay-scraper scraper and, more specifically, the listings collection. You can add as many scrapers and as many collections here as you want, but keep in mind that the same QA validation rules will be applied to all of them. Basically, whatever scrapers and collections you add should have similar output. If you have another scraper that has different output and different validation rules, you will need to create a separate QA scraper.

7.1.2 Step 2: Seeding

Now, let's create a seeder script. This is where we will iterate through the scrapers that we just defined and seed them to be fetched by DataHen. In this example we will just seed one scraper.

Create the seeder directory:

```
mkdir seeder
```

Next, let's create a ruby script called seeder.rb in the seeder directory with the following content:

```
config_path = File.expand_path('dh_easy.yaml', Dir.pwd)
config = YAML.load(File.open(config_path))['qa']
config['scrapers'].each do |scraper_name, collections|
  if collections
    pages << {
      url: "https://fetchtest.datahen.com/?scraper=#{scraper_name}",
      method: "GET",
      page_type: "qa",
      vars: {
        scraper_name: scraper_name,
        collections: collections
      }
    }
  end
end
```

This will create a page on DataHen with the scraper (ebay-scraper) and collection (listings) that we want to validate saved in the vars. We will eventually create a parser that will send these values to the dh_easy-qa gem which will load the relevant data from DataHen and perform QA on it using rules that we will eventually define as well.

Let's go back into the root directory of the project

```
$ cd ..
$ ls -alth
total 0
drwxr-xr-x  10 johndoe  staff   320B 26 Nov 16:19 .git
drwxr-xr-x   3 johndoe  staff    96B 26 Nov 16:15 seeder
drwxr-xr-x   1 johndoe  staff    57B 26 Nov 16:15 Gemfile
drwxr-xr-x   4 johndoe  staff   128B 26 Nov 16:15 .
drwxr-xr-x  10 johndoe  staff   320B 26 Nov 15:59 ..
```

Now that we've created the seeder script, let's see if there are any syntax error in it by trying the seeder script.

```
$ hen seeder try ebay-example-qa seeder/seeder.rb
Trying seeder script
=========== Seeding Executed ===========
----------------------------------------
Would have saved 1 out of 1 Pages
[
  {
    "url": "https://fetchtest.datahen.com/?scraper=ebay-scraper",
    "method": "GET",
    "page_type": "qa",
    "force_fetch": true,
    "vars": {
      "scraper_name": "ebay-scraper",
      "collections": [
        "listings"
      ]
    }
  }
]
```

Looks like our seeder script test was successful. Let's now create the config file to tell DataHen about our seeder. Create a config.yaml file in the root project directory with the following content:

```
seeder:
 file: ./seeder/seeder.rb
 disabled: false # Optional. Set it to true if you want to disable execution of this␣
↪file
```

The config above simply tells DataHen where the seeder file is, so that it can be executed.

Let's now commit our files with git and push them to Github. Add all of the current files with the following commands:

```
$ git add .
$ git commit -m 'create inital qa files'
[master (root-commit) 7632be0] create initial qa files
 1 file changed, 5 insertions(+)
 create mode 100644 seeder/seeder.rb
 create mode 100644 Gemfile
 create mode 100644 Gemfile.lock
 create mode 100644 dh_easy.yaml
 create mode 100644 config.yaml
```

Next, let's push it to an online git repository provider. In this case let's push this to Github. In the example below it is using our git repository, you should push to your own repository.

```
$ git remote add origin https://github.com/DataHenOfficial/ebay-scraper-qa.git
$ git push -u origin master
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 382 bytes | 382.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/DataHenOfficial/ebay-scraper-qa/pull/new/master
remote:
To https://github.com/DataHenOfficial/ebay-scraper-qa.git
```

(continues on next page)

(continued from previous page)

```
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Congratulations, you've successfully seeded the scraper collections we want to validate with QA and are ready to define some validation rules, which we will do in the next step.

7.1.3 Step 3: QA Validation Rules

Now that we have seeded a page to DataHen with info about the scraper that we want to perform QA on, we can define some validation rules. Add some lines to the dh_easy.yaml file so that it looks like the following:

```
qa:
  scrapers:
    ebay-example:
      - listings
  individual_validations:
    url:
      required: true
      type: Url
    title:
      required: true
      type: String
```

Basically this will tell the QA gem to perform the following validations on the listings collection of our ebay-example scraper. It will make sure the url and the title are both present and it will make sure that the url output is actually a url and that the title output is a string. If any of these validations fail, the failure will be returned in a summary and the specific listing will be returned with the corresponding failure.

Next we need to create a parser, which will parse the page that has our scraper (ebay-example) and collection (listings) info saved in vars, and use the dh_easy-qa gem to perform QA. In our seeder we set the page_type to qa, so lets create a parser named qa.rb inside a folder named parsers.

First lets create the parsers directory:

```
mkdir parsers
```

Next create a file called qa.rb inside this parsers directory with the following lines:

```
require 'dh_easy/qa'
DhEasy::Qa::Validator.new.validate_internal(page['vars'], outputs)
```

Here we are telling the QA gem to validate internal scrapers on DataHen and are passing the details of these scrapers inside the vars. We also pass the outputs array which is a special reserved word in DataHen that is an array of job output to be saved. This way the QA gem will be able to save the QA output to DataHen for you to see.

Let's retrieve the GID of the page that we seeded earlier so we can try it out locally. Run the following command in the project root directory.

```
hen scraper page list ebay-example-qa
[
 {
  "gid": "fetchtest.datahen.com-1767f1fa6b7302b4a618b16b470fc1d2",
  "job_id": 9793,
  "job_status": "active",
  "status": "parsing_failed",
```

(continues on next page)

(continued from previous page)

```
  "fetch_type": "standard",
  "page_type": "qa",
  "priority": 0,
  "method": "GET",
  "url": "https://fetchtest.datahen.com/?scraper=ebay-example",
  "effective_url": "https://fetchtest.datahen.com/?scraper=ebay-example",
  "headers": null,
  "cookie": null,
  "body": null,
  "created_at": "2019-08-09T21:44:18.709737Z",
  "no_redirect": false,
  "ua_type": "desktop",
  "freshness": "2019-08-09T21:44:18.735754Z",
  "fresh": true,
  "parsing_at": null,
  "parsing_failed_at": "2019-08-09T22:05:30.684121Z",
  "parsed_at": null,
  "parsing_try_count": 3,
  "parsing_fail_count": 3,
  "fetched_at": "2019-08-09T21:45:10.312099Z",
  "fetching_try_count": 1,
  "to_fetch": "2019-08-09T21:44:18.73264Z",
  "fetched_from": "web",
  "response_checksum": "9d650deb8d3fd908de452f27e148293d",
  "response_status": "200 OK",
  "response_status_code": 200,
  "response_proto": "HTTP/1.1",
  "content_type": "text/html; charset=utf-8",
  "content_size": 555,
  "vars": {
   "collections": [
    "listings"
   ],
   "scraper_name": "ebay-example"
  },
  "failed_response_status_code": null,
  "failed_response_headers": null,
  "failed_response_cookie": null,
  "failed_effective_url": null,
  "failed_at": null,
  "failed_content_type": null,
  "force_fetch": false
 }
]
```

We can see the scraper name and collection are both present in "vars," but what we are interested in is the gid which will look something like, "fetchtest.datahen.com-1767f1fa6b7302b4a618b16b470fc1d2." We can use the gid to try out our parser, which will perform the QA, on this page.

Run the following command, replacing the <gid> part with your gid value:

```
hen parser try ebay-example-qa parsers/qa.rb <gid>
```

The output should look something like:

```
Trying parser script
getting Job Page
```

(continues on next page)

(continued from previous page)

```
1
2
validating scraper: ebay-example
Validating collection: listings
data count 42
=========== Parsing Executed ===========
----------------------------------------
Would have saved 1 out of 1 Outputs
[
  {
    "pass": "true",
    "_collection": "ebay-example_listings_summary",
    "total_items": 42
  }
]
```

This means that our validation rules in dh_easy.yaml have passed for each of the 42 items. This also means that you have successfully performed basic QA on your scraper! We will look at more advanced settings in the next section.

7.1.4 Additional Validation Rules

These are examples of all the available validation rules. You use them by adding them to dh_easy.yaml nested under 'individual_validations.' For example, here is a validation rule that makes sure a field named, "Title" is present, is a string, and has a length of 10.

```
qa:
  individual_validations:
    Title:
      required: true
      type: String
      length: 10
```

Here are all the possible validation rules and values:

```
length: 5
```

Validates the length of a field value. The field value can be an Integer, Float, or a String.

```
type: String
```

Validates that a value is a String.

```
type: Integer
```

Validates that a value is an Integer. The value can be a number in a String. Examples that would pass: 10, '10', '1,000'.

```
type: Float
```

Validates that a value is a Float (a number with a decimal point). The value can be a number with decimal points in a String. Examples that would pass: 2.0, '3.14', '99.99'.

```
type: Date
format: '%d-%m-%Y'
```

Validates that a value is a date. A format is required using Ruby strftime.

```
type: Url
```

Validates that a value is a valid url.

```
value:
  equal: 'test'
```

You can also add validations that validate the value of a field itself. For example, the above validation will validate that a field is equal to the string, 'test'.

```
value:
  less_than: 5
```

You can also verify that a field is less than something.

```
value:
  greater_than: 100
```

You can also verify that a field is more than something.

```
value:
  regex: "^(Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday)$"
```

Validates the value of a field using a regular expression. For example, you could validate that a value is a phone number or a day of the week like the above example. Regex uses case ignore by default.

```
title:
  value:
    equal: 'Test title'
    if:
      search_input:
        value:
          equal: 'Search'
```

You can also implement conditions on value validations. For instance, the above example validates that the value of a field named, 'title' has a value equal to 'Test title' only if the value of the field named, 'search_input' has a value equal to, 'Search.' If statements currently support value checks with the same options as a normal value check (less_than, greater_than, regex, and equal).

```
title:
  required: true
  if:
    search_input:
      value:
        regex: '(One|Two|Three)'
```

You can also implement an if condition on 'required.' The above example will only check if the 'title' field is required if the field named, 'search_input' has a value equal to: 'One,' 'Two,' or 'Three.'

```
title:
  required: true
  if:
    and:
      -
       field1:
         value:
           regex: '(One|Two)'
      -
       field2:
         value:
           less_than: 100
```

If conditions can also take 'and' and 'or' operators. The above example shows a validation that will only check if the 'title' field is required if the field named, 'field1' has a value equal to: 'One' or 'Two' and the field named, 'field2' has a value that is less than 100.

7.1.5 Group Validations

In addition to these individual validations you can also perform more complicated validations on the data as a whole. For example, you may want to ensure that a specific field has ranked values and are in order. To add group validations create a file named group_validations.rb in your QA scraper root directory with the following (data is an array of the items you are performing QA on):

```
module GroupValidations
  def validate_count
    errors[:group_validation] = { failure: 'count' } if data.count > 100
  end
end
```

This example would create an error if the total number of items is greater than 100. Let's look at another another example.

```
module GroupValidations
  def validate_count
    errors[:group_validation] = { failure: 'count' } if data.count > 100
  end

  def validate_unique_ids
    ids = data.collect{|item| item['id'] }
    errors[:group_validation] = { failure: 'unique_ids' } if ids.uniq.count != ids.
↪count
  end
end
```

This would add another validation that checks to make sure all item 'id' values are unique. You can edit these examples to create your own.

7.1.6 Thresholds

Thresholds are useful if you want to suppress errors based on frequency. You can suppress errors on the basis of the number of errors relative to the number of items you are performing QA on. The threshold itself is a number between 0 and 1 where 1 means that if any error occurs, the error will show. A threshold of 0 means we are ignoring all errors. There are multiple ways you can set a threshold which we will go through below.

We can set a threshold on a per field basis which will apply to all scrapers. This can be done by adding "threshold" to the dh_easy.yaml file to a specific field just like a rule. For example, the following will add a threshold that will only show errors on the "Title" field for every scraper if the occurance rate is above 60%.

```
qa:
  individual_validations:
    Title:
      threshold: 0.6
      required: true
      type: String
      length: 10
```

We can also set thresholds on a per field and a per scraper scraper basis at the same time. Using the "Title" field example, this means you could have a threshold of 0.6 for the Title on one scraper and have 1.0 for another scraper. In order to implement different thresholds for individual scrapers you can create a file called thresholds.yaml in your scraper root directory. Here is an example of a thresholds.yaml file that would apply different thresholds for a scraper named ebay1 and a scraper named ebay2.

```
---
ebay1:
   Title:
     threshold: 0.6
     required: true
     type: String
ebay2:
   Title:
     threshold: 1.0
     required: true
     type: String
```

7.1.7 External sources

In addition to performing validation on scrapers that run on DataHen (internal sources) you can also perform validation on external sources. For example, if you have a scraper that runs somewhere else, you can validate it by ingesting a json endpoint. Here is an example seeder for an external source:

```
pages << {
   url: "http://dummy.restapiexample.com/api/v1/employees",
   method: "GET",
   force_fetch: true,
   freshness: Time.now.iso8601,
   vars: {
     collection_id: "employees-1"
   }
}
```

This seeder could be expanded to seeding multiple endpointpoints by loading a YAML file and iterating through like in Step 2 above. After seeding our external json endpoint we can now write a parser such as the following:

```
require 'typhoeus'
require 'json'
require 'dh_easy/qa'

collection_name = page['vars']['collection_id']
json = JSON.parse(content)
qa = DhEasy::Qa::Validator.new(json, {})
qa.validate_external(outputs, collection_name)
```

We can also implement thresholds with external sources by loading a thresholds yaml and passing it into the validator options. We can update our parser so that it looks like the following:

```
require 'typhoeus'
require 'json'
require 'yaml'
require 'dh_easy/qa'

collection_name = page['vars']['collection_id']
file_path = File.expand_path('thresholds.yaml', Dir.pwd)
thresholds = YAML.load(File.open(file_path))
options = { 'thresholds' => thresholds[collection_name] }

json = JSON.parse(content)
qa = DhEasy::Qa::Validator.new(json, options)
qa.validate_external(outputs, collection_name)
```

CHAPTER 8

How-Tos

The following are a list of commonly occuring DataHen use cases and how to do them. It is useful to skim through all the how-tos so that in the future, if you come across any similar need, you can refer back to this section.

8.1 Setting a scraper's scheduler

You can schedule a scraper's scheduler in as granular detail as you want. However, we only support granularity down to the Minute. We currently support the CRON syntax for scheduling. You can set the 'schedule' field or the 'timezone' to specify the timezone when the job will be started. If timezone is not specified, it will default to "UTC". Timezone values are IANA format. Please see the list of available timezones. The scheduler also has the option to cancel current job, anytime it starts a new one. By default a scraper's scheduler does not start a new job if there is an already existing job that is active.

```
$ hen scraper create <scraper_name> <git_repo> --schedule "0 1 * * * *" --timezone
↪"America/Toronto" --cancel-current-job
$ hen scraper update <scraper_name> --schedule "0 1 * * * *" --timezone "America/
↪Toronto"
```

The following are allowed CRON values:

Field Name	Mandatory?	Allowed Values	Allowed Special Characters
Minutes	Yes	0-59	* / , -
Hours	Yes	0-23	* / , -
Day of month	Yes	1-31	* / , - L W
Month	Yes	1-12 or JAN-DEC	* / , -
Day of week	Yes	0-6 or SUN-SAT	* / , - L #
Year	No	1970–2099	* / , -

Asterisk (*)

The asterisk indicates that the cron expression matches for all values of the field. E.g., using an asterisk in the 4th field (month) indicates every month.

Slash (/)

Slashes describe increments of ranges. For example 3-59/15 in the minute field indicate the third minute of the hour and every 15 minutes thereafter. The form */… is equivalent to the form "first-last/…", that is, an increment over the largest possible range of the field.

Comma (,)

Commas are used to separate items of a list. For example, using MON,WED,FRI in the 5th field (day of week) means Mondays, Wednesdays and Fridays.

Hyphen (-)

Hyphens define ranges. For example, 2000-2010 indicates every year between 2000 and 2010 AD, inclusive.

L

L stands for "last". When used in the day-of-week field, it allows you to specify constructs such as "the last Friday" (5L) of a given month. In the day-of-month field, it specifies the last day of the month.

W

The W character is allowed for the day-of-month field. This character is used to specify the business day (Monday-Friday) nearest the given day. As an example, if you were to specify 15W as the value for the day-of-month field, the meaning is: "the nearest business day to the 15th of the month."

So, if the 15th is a Saturday, the trigger fires on Friday the 14th. If the 15th is a Sunday, the trigger fires on Monday the 16th. If the 15th is a Tuesday, then it fires on Tuesday the 15th. However if you specify 1W as the value for day-of-month, and the 1st is a Saturday, the trigger fires on Monday the 3rd, as it does not 'jump' over the boundary of a month's days.

The W character can be specified only when the day-of-month is a single day, not a range or list of days.

The W character can also be combined with L, i.e. LW to mean "the last business day of the month."

Hash (#)

is allowed for the day-of-week field, and must be followed by a number between one and five. It allows you to specify constructs such as "the second Friday" of a given month.

8.2 Changing a Scraper's or a Job's Proxy Type

We support many types of proxies to use:

Proxy Type	Description
standard	The standard rotating proxy that gets randomly used per request. This is the default.

```
$ hen scraper update <scraper_name> --proxy-type sticky1
```

Keep in mind that the above will only take effect when a new scrape job is created.

To change a proxy of an existing job, first cancel the job, and then change the proxy_type, and then resume the job:

```
$ hen scraper job cancel <scraper_name>
$ hen scraper job update <scraper_name> --proxy-type sticky1
$ hen scraper job resume <scraper_name>
```

8.3 Setting a specific ruby version

By default our ruby version that we use is 2.4.4, however if you want to specify a different ruby version you can do so by creating a .ruby-version file on the root of your project directory.

NOTE: we currently only allow the following ruby versions:

- 2.4.4
- 2.5.3
- If you need a specific version other than these, please let us know

8.4 Setting a specific Ruby Gem

To add dependency to your code, we use Bundler. Simply create a Gemfile on the root of your project directory.

```
$ echo "gem 'roo', '~> 2.7.1'" > Gemfile
$ bundle install # this will create a Gemfile.lock
$ ls -alth | grep Gemfile
total 32
-rw-r--r--   1 johndoe  staff    22B 19 Dec 23:43 Gemfile
-rw-r--r--   1 johndoe  staff   286B 19 Dec 22:07 Gemfile.lock
$ git add . # and then you should commit the whole thing into Git repo
$ git commit -m 'added Gemfile'
$ git push origin
```

8.5 Changing a Scraper's Standard worker count

The more workers you use on your scraper, the faster your scraper will be. You can use the command line to change a scraper's worker count:

```
$ hen scraper update <scraper_name> --workers N
```

Keep in mind that this will only take effect when a new scrape job is created.

8.6 Changing a Scraper's Browser worker count

The more workers you use on your scraper, the faster your scraper will be. You can use the command line to change a scraper's worker count:

```
$ hen scraper update <scraper_name> --browsers N
```

NOTE: Keep in mind that this will only take effect when a new scrape job is created.

8.7 Changing an existing scrape job's worker count

You can use the command line to change a scraper job's worker count:

```
$ hen scraper job update <scraper_name> --workers N --browsers N
```

This will only take effect if you cancel, and resume the scrape job again:

```
$ hen scraper job cancel <scraper_name> # cancel first
$ hen scraper job resume <scraper_name> # then resume
```

8.8 Enqueueing a page to Browser Fetcher's queue

You can enqueue a page like so in your script. The following will enqueue a headless browser:

```
pages << {
  url: "http://test.com",
  fetch_type: "browser" # This will enqueue headless browser
}
```

Or use the command line:

```
$ hen scraper page add <scraper_name> <url> --fetch-type browser
```

You can enqueue a page like so in your script. The following will enqueue a full browser (non-headless):

```
pages << {
  url: "http://test.com",
  fetch_type: "fullbrowser" # This will enqueue headless browser
}
```

Or use the command line:

```
$ hen scraper page add <scraper_name> <url> --fetch-type fullbrowser
```

8.9 Setting fetch priority to a Job Page

The following will enqueue a higher priority page. NOTE: You can only create a page that has priority, not update an existing page with a new priority value on the script. Also, updating a priority only works via the command line tool.

```
pages << {
  url: "http://test.com",
  priority: 1 # defaults to 0. Higher numbers means will get fetched sooner
}
```

Or use the command line:

```
$ hen scraper page add <scraper_name> <url> --priority N
$ hen scraper page update <job> <gid> --priority N
```

8.10 Setting a user-agent-type of a Job Page

You can enqueue a page like so in your script:

```
pages << {
  url: "http://test.com",
  ua_type: "desktop" # defaults to desktop, other available is mobile.
}
```

Or use the command line:

```
$ hen scraper page add <scraper_name> <url> --ua-type mobile
```

8.11 Setting the request method of a Job Page

You can enqueue a page like so in your script:

```
pages << {
  url: "http://test.com",
  method: "POST" # defaults to GET.
}
```

Or use the command line:

```
$ hen scraper page add <scraper_name> <url> --method GET
```

8.12 Setting the request headers of a Job Page

You can enqueue a page like so in your script:

```
pages << {
  url: "http://test.com",
  headers: {"Cookie": "name=value; name2=value2; name3=value3"} # set this
}
```

Or use the command line:

```
$ hen scraper page add <scraper_name> <url> --headers '{"Cookie": "name=value;␣
↪name2=value2; name3=value3"}'
```

8.13 Setting the request body of a Job Page

You can enqueue a page like so in your script:

```
pages << {
  url: "http://test.com",
  body: "your request body here" # set this field
}
```

Or use the command line:

```
$ hen scraper page add <scraper_name> <url> --body 'your request body here'
```

8.14 Setting the page_type of a Job Page

You can enqueue a page like so in your script:

```
pages << {
  url: "http://test.com",
  page_type: "page_type_here" # set this field
}
```

Or use the command line:

```
$ hen scraper page add <scraper_name> <url> --page-type page_type_here
```

8.15 Reset a Job Page

You can reset a scrape-job page's parsing and fetching from the command line:

```
$ hen scraper page reset <scraper_name> <gid>
```

You can also reset a page from any parser or seeder script by setting the *reset* field to true while enqueueing it, like so:

```
pages << {
  url: "http://test.com",
  reset: true # set this field
}
```

8.16 Handling cookies

There are two ways to handle cookies in DataHen, at a lower level via the Request and Response Headers, or at a higher level via the Cookie Jar.

8.16.1 Low level cookie handling using Request/Response Headers

To handle cookie at a lower level, you can set the "cookie" on the request header:

```
pages << {
  url: "http://test.com",
  headers: {"Cookie": "name=value; name2=value2; name3=value3"},
}
```

You can also read cookies by reading the “Set-Cookie” response headers:

```
page['response_headers']['Set-Cookie']
```

8.16.2 High level cookie handling using the Cookie Jar

To handle cookie at a higher level, you can set the “cookie” field directly onto the page, and it will be saved onto the Cookie Jar during that request.

```
pages << {
  url: "http://test.com",
  cookie: "name=value; name2=value2; name3=value3",
}
```

You can also do so from the command line:

```
$ hen scraper page add <scraper_name> <url> --cookie "name=value; name2=value2"
```

You can then read the cookie from the cookiejar by:

```
page['response_cookie']
```

This method above is reading from the cookiejar. This is especially useful when a cookie is set by the target-server during redirection.

8.17 Force Fetching a specific unfresh page

To enqueue a page and have it force fetch, you need to set freshness field, and force_fetch field. Freshness should only be now, or in the past. It cannot be in the future. Basically it is “how much time ago, that you consider this page as fresh” One thing to keep in mind, that this only resets the page fetch, it does nothing to your parsing of pages, whether the parser has executed or not. In your parser script you can do the following:

```
pages << {
  url: "http://test.com",
  freshness: "2018-12-12T13:59:29.91741Z", # has to be this string format
  force_fetch: true
}
```

You can do this to find one output result or use the command line to query an output:

```
$ hen scraper page add <scraper_name> <url> --page-type page_type_here --force-fetch -
↪-freshness "2018-12-12T13:59:29.91741Z"
```

8.18 Handling JavaScript

To do javascript rendering, please use the Browser Fetcher. First you need to add a browser worker onto your scraper:

```
$ hen scraper update <scraper_name> --browsers 1
```

Next, for every page that you add, you need to specify the correct fetch_type:

```
$ hen scraper page add <scraper_name> <url> --fetch-type browser
```

Or in the script, by doing the following:

```
pages << {
  url: "http://test.com",
  fetch_type: "browser"
}
```

8.19 Browser display

We support display size configuration within Browser Fetcher having 1366x768 as default size. This feature is quite useful when interacting with responsive websites and taking screenshots. Only *browser* and *fullbrowser* fetch types support this feature.

First you need to add a browser worker onto your scraper:

```
$ hen scraper update <scraper_name> --browsers 1
```

This example shows you how to change the browser display size to 1920x1080:

```
pages << {
  "url": "https://www.datahen.com",
  "page_type": "homepage",
  "fetch_type": "browser",
  "display": {
    "width": 1920,
    "height": 1080
  }
}
```

8.20 Browser interaction

We support browser interaction through Puppeteer and Browser Fetcher. Only *browser* and *fullbrowser* fetch types support this feature.

We fully support JS puppeteer's page object and provide a predefined *sleep(miliseconds)* async function to allow easy browser interaction and actions.

First you need to add a browser worker onto your scraper:

```
$ hen scraper update <scraper_name> --browsers 1
```

Next you will need to add your puppeteer javascript code to interact with your browser fetch when enqueuing your page inside your seeder or parser scripts.

This example shows you how to click the first footer link and wait 3 seconds after the page has load:

```
pages << {
  "url": "https://www.datahen.com",
  "page_type": "footer_page",
  "fetch_type": "browser",
  "driver": {
```

(continues on next page)

(continued from previous page)

```
    "code": "await page.click('footer ul > li > a'); await sleep(3000);"
  }
}
```

Notice that modifying your driver code will generate the same GID, to change this, assign driver's *name* attribute.

8.20.1 Enqueue same page twice with diffeent code

Sometimes, you will need to scrape the same page more than one time but interact with it on a different way, therefore, *driver.code* attribute alone will generate same GID everytime when using the same page configuration.

To fix this, use *driver.name* attribute as a unique identifier to your *driver.code* and change the GID.

This example shows you how to enqueue the same page twice with different browser interaction by using *name* attribute, notice that each enqueued page will now generate it's own unique GID:

```
pages << {
  "url": "https://www.datahen.com",
  "page_type": "footer_page",
  "fetch_type": "browser",
  "driver": {
    "name": "click first footer link"
    "code": "await page.click('footer ul > li > a'); await sleep(3000);"
  }
}

pages << {
  "url": "https://www.datahen.com",
  "page_type": "footer_page",
  "fetch_type": "browser",
  "driver": {
    "name": "click second footer link"
    "code": "await page.click('footer ul > li + li > a'); await sleep(3000);"
  }
}
```

8.20.2 Change browser fetch behavior

We have a 30 seconds default timeout on each browser fetch therefore, you might find that some pages having timeout on Browser Fetcher because of heavy resources taking too much time to load or maybe a heavy loading API response, that will likely cause your pages to fail.

To fix this, change your page browser timeout to be as long as you need by using *driver.goto_options*. This example shows you how to increase your page browser timeout to 50 seconds:

```
pages << {
  "url": "https://www.datahen.com",
  "page_type": "homepage",
  "fetch_type": "browser",
  "driver": {
    "goto_options": {
      "timeout": 50000
    }
  }
}
```

driver.goto_options attribute fully supports puppeteer's *page.goto options* param, you can learn more about it here.

8.20.3 Dealing with responsive designs

Response designs are quite common along websites, which makes it a common problem when comes to browser interaction click actions on elements that would be hidden on smaller or bigger screen sizes.

This example shows you how to use *display* options to set your browser display size to mobile portrait and then click on a menu option from a response website:

```
pages << {
  "url": "https://www.datahen.com",
  "page_type": "mobile_blog",
  "fetch_type": "browser",
  "display": {
    "width": 320,
    "height": 480
  }
  "driver": {
    "code": "await page.click('hamburger-toggle'); await sleep(3000); page.click('.
↪menu-horizontal > li + li + li+ li + li + li > a')"
  }
}
```

8.20.4 Dealing with infinite load timeouts

There are some weird scenarios on which a website will just never finish loading becuase a buggy resource or a never ending JS script loop, that will trigger a timeout no matter how much you wait.

A good way to deal with these weird scenarios is to use puppeteer's goto option *domcontentloaded* and our predefined sleep async function. The next example shows you how to combine these two options into a working solution by manually waiting 3 seconds for the page to load:

```
pages << {
  "url": "https://www.datahen.com",
  "page_type": "homepage",
  "fetch_type": "browser",
  "driver": {
    "code": "await sleep(3000);",
    "goto_options": {
      "waitUntil": "domcontentloaded"
    }
  }
}
```

8.21 Taking screenshots

We support browser screenshots within Browser Fetcher by enabling *screenshot.take_screenshot* attirbute. It is important to note that taking a screenshot will replace the page *content* with the screenshot binary contents. Only *browser* and *fullbrowser* fetch types support this feature.

First you need to add a browser worker onto your scraper:

```
$ hen scraper update <scraper_name> --browsers 1
```

Next you need to enqueue your page with *screenshot.take_screenshot* attribute enabled. This example shows you how to take a screenshot:

```
# ./seeder/seeder.rb
pages << {
  "url": "https://www.datahen.com",
  "page_type": "homepage",
  "fetch_type": "browser",
  "screenshot": {
    "take_screenshot": true,
    "options": {
     "fullPage": false,
     "type": "jpeg",
     "quality": 75
   }
  }
}
```

This will replace the page's html source code at "content" variable with the screenshot binary.

This example shows you how to save the screenshot to an AWS S3 bucket, but first, let's create our prerequisites, *Gemfile* and *config.yml* files:

```
# Gemfile
source 'https://rubygems.org'
gem 'datahen'
gem 'aws-sdk-s3'
```

```
# config.yml
seeder:
  file: ./seeder/seeder.rb
  disabled: false
parsers:
  - file: ./parser/upload_to_s3.rb
    page_type: my_screenshot
    disabled: false
```

Now we can upload our screenshot to AWS S3 to our *my_bucket* bucket as *my_screenshot.jpeg*:

```
# ./parser/upload_to_s3.rb
require 'aws-sdk-s3'

s3 = Aws::S3::Resource.new()
obj = s3.bucket('your_bucket').object('my_screenshot.jpeg')
obj.put(body: content)
```

8.21.1 Screenshot options

We support all options from puppeteer's *page.screenshot options* params other than *path* and *encoding* due internal handling. You can learn more about it here.

This example shows you how to take a full page screenshot as *JPEG*:

```
pages << {
  "url": "https://www.datahen.com",
  "page_type": "homepage",
  "fetch_type": "browser",
  "screenshot": {
    "take_screenshot": true,
    "options": {
     "fullPage": true,
     "type": "jpeg",
     "quality": 75
   }
  }
}
```

And this example shows you how to take a 800x600 display size screenshot as *PNG*:

```
pages << {
  "url": "https://www.datahen.com",
  "page_type": "homepage",
  "fetch_type": "browser",
  "display": {
    "width": 800,
    "height": 600
  }
  "screenshot": {
    "take_screenshot": true,
    "options": {
     "fullPage": false,
     "type": "png"
   }
  }
}
```

Notice that *PNG* screenshots doesn't support *screenshot.quality* attribute, more information about it here.

8.21.2 Screenshots and browser interaction

Screenshots and Browser Fetch interaction are compatible, so you can use both to interact with your page before taking a screenshot.

This example shows you how to take a screenshot of *duckduckgo.com* homepage after showing it's side menu at 1920x1080:

```
pages << {
  "url": "https://www.datahen.com",
  "page_type": "homepage",
  "fetch_type": "browser",
  "display": {
    "width": 1920,
    "height": 1080
  }
  "driver": {
    "code": "page.click('.js-side-menu-open'); await sleep(3000);"
  },
  "screenshot": {
    "take_screenshot": true,
```

(continues on next page)

(continued from previous page)

```
    "options": {
      "fullPage": false,
      "type": "png"
    }
  }
}
```

8.22 Doing dry-run of your script locally

Using the *try* command will allow you dry-run a parser or a seeder script locally. How it works is, it downloads necessary data from the DataHen cloud, and then executes your script locally, but it does not upload any data back to the DataHen Cloud.

```
$ hen parser try ebay parsers/details.rb
$ hen seeder try ebay seeder/seeder.rb
```

8.23 Executing your script locally, and uploading to DataHen

Using the *exec* command will allow you execute a parser or a seeder script locally and upload the result to the DataHen cloud. It works by downloading the necessary data from the DataHen cloud, and executes it locally. When done it will upload the resulting output and pages back onto the DataHen cloud.

```
$ hen parser exec <scraper_name> <parser_file> <gid>
$ hen seeder exec <scraper_name> <seeder_file>
```

The *exec* command is really useful to do end-to-end testing on your script, to ensure that not only the execution works, but also if it properly uploads the resulting data to the DataHen cloud. Any errors that are generated during the exec command, will be logged onto the DataHen cloud's log, so it is accessible in the following way

```
$  hen scraper log <scraper_name>
$  hen scraper page log <scraper_name> <gid>
```

Once you've successfully executed the command locally using *exec* you can check your stats, and collection lists and outputs using the command

```
$ hen scraper stats <scraper_name>
$ hen scraper output collection <scraper_name>
$ hen scraper output list <scraper_name> --collection <collection_name>
```

8.24 Querying scraper outputs

We currently support the ability to query a scraper outputs based on arbitrary JSON key. Only exact matches are currently supported. In your parser script you can do the following to find many output results:

```
# find_outputs(collection='default', query={}, page=1, per_page=30)
# will return an array of output records
records = find_outputs('foo_collection', {"_id":"123"}, 1, 500)
```

Or you can do this to find one output result:

```
# find_output(collection='default', query={})
# will return one output record
record = find_output('foo_collection', {"_id":"123"})
```

Or use the command line, to query an output:

```
$ hen scraper output list <scraper_name> --collection home --query '{"_id":"123"}'
```

You can also query outputs from another scraper or job: To find output from another job, do the following:

```
records = find_outputs('foo_collection', {"_id":"123"}, 1, 500, job_id: 1234}
```

To find output from another scraper, do the following:

```
records = find_outputs('foo_collection', {"_id":"123"}, 1, 500, scraper_name:'my_
↪scraper'}
```

8.25 Restart a scraping job

To restart a job, you need to cancel an existing job first, then start a new one:

```
$ hen scraper job cancel <scraper_name>
$ hen scraper start <scraper_name>
```

8.26 Setting Environment Variables and Secrets on your account.

You can set any environment variables and secrets in your account, that you can then use in any of your scrapers.

There are similarities between environment variables and secrets, that they are equally accessable on any of your seeder, parser, finisher scripts. The difference is, secrets are encrypted.

Secrets are useful to store things such as, passwords, or connection strings if you need to connect to a database, etc.

Another benefit of using environment variables and secret is so that you don't have to store any values in the Git repository. This will make your code more secure and more reusable.

This example scraper shows usage of environment variables.

There are three steps that you need to do in order to use environment variables and secrets:

8.26.1 1. Set the environment variable or secrets on your account.

To set an environment variable using command line:

```
$ hen var set <var_name> <value>
```

To set a secret environment variable using command line:

```
$ hen var set <var_name> <value> --secret
```

8.26.2 2. Change your config.yaml to use the variables or secrets.

Add the following to your config.yaml file.

```
env_vars:
 - name: foo
   global_name: bar # Optional. If specified, refers to your account's environment␣
↪variable of this name.
   disabled: false # Optional
 - name: baz
   default: bazvalue
```

In the example above, this will search for your account's environment variable of `bar` and then make it available to your script as `ENV['foo']`. The above example also will search for `baz` variable on your account, and make it available to your script as `ENV['baz']`.

IMPORTANT: The name of the env var must be the same as the env var that you have specified in your account in step 1. If You intend to use a different variable name in the scraper vs in the account, use `global_name`.

8.26.3 3. Access the environment variables and secrets in your script.

Once you've done step 1 and 2 above, you can then access the environment variables or secrets from any of your seeder, parser, finisher scripts, by doing so:

```
ENV['your_env_var_here']
```

8.27 Setting Input Variables and Secrets on your scraper and scrape job.

You can set any input variables and secrets on your scraper, similar to how you use environment variables.

There are similarities between input variables and secrets, that they are equally accessable on any of your seeder, parser, finisher scripts. The difference is, secrets are encrypted.

Secrets are useful to store things such as, passwords, or connection strings if you need to connect to a database, etc.

Another benefit of using input variables and secret is so that you don't have to store any values in the Git repository. This will make your code more secure and more reusable.

When you've specified your input variables on your scraper, any scrape jobs will then be started with the input variables that are taken from your scraper's input variables.

This example scraper shows usage of input variables.

There are three steps that you need to do in order to use input variables and secrets:

8.27.1 1. Set the input variable or secrets on your scraper.

To set an input variable on a scraper using command line:

```
$ hen scraper var set <var_name> <value>
```

To set a secret input variable on a scraper using command line:

```
$ hen scraper var set <var_name> <value> --secret
```

To set an input variable on a scrape job using command line:

```
$ hen scraper job var set <var_name> <value>
```

IMPORTANT: For this to take effect. You must pause and resume the job

To set a secret input variable on a scraper job using command line:

```
$ hen scraper job var set <var_name> <value> --secret
```

IMPORTANT: For this to take effect. You must pause and resume the job

8.27.2 2. Change your config.yaml to use the variables or secrets.

Add the following to your config.yaml file.

```
input_vars:
 - name: starting_url
   title: Starting URL # Optional
   description: Enter the starting URL for the scraper to run # optional
   default: https://www.ebay.com/sch/i.html?_nkw=macbooks # optional.
   type: text # Available values include: string, text, secret, date, datetime. This␣
↪will display the appropriate input on the form.
   required: true # Optional. This will make the input field in the form, required
   disabled: false # Optional
 - name: baz
```

In the example above, this will search for your scrape job's input variable of `starting_url` and then make it available to your script as `ENV['starting_url']`. The above example also will search for `baz` variable on your scrape job, and make it available to your script as `ENV['baz']`.

8.27.3 3. Access the input variables and secrets in your script.

Once you've done step 1 and 2 above, you can then access the input variables or secrets from any of your seeder, parser, finisher scripts, by doing so:

```
ENV['your_input_var_here']
```

8.28 Using a custom docker image for the scraper

We support docker image where the scraper will run on. What this means, is that you can install any dependencies that you'd like on it. Please let the DataHen support know so that this can be created for you.

IMPORTANT: Only docker images that are compatible with DataHen can be run. Please contact us for more info.

Our base Docker image is based on Alpine 3.7:

```
FROM alpine:3.7
```

So, if you want a package to be installed, make sure that it builds correctly on your local machine first.

Once correctly built, please let us know what dockerfile commands to add to the custom image. The following format would be preferable:

```
RUN apk add --update libreoffice
```

Once we have built the image for you, you can use this custom image by modifying your config.yaml file and include the following line:

```
scraper_image: <url-to-your-docker-image>
```

When you have modified this and deploy this, you need to restart your job.

8.29 How to use shared code libraries from other Git repositories using Git Submodule

Sometimes you want to have a scraper that has a shared list of libraries that are used by other scrapers in other Git repositories. Luckily DataHen supports Git Submodules, which enables this scenario.

You simply just deploy a scraper as usual, and DataHen will take care of initating and checking out the submodules recursively.

This is the documentation on Git Submodules that shows the usage in depth.

This example scraper shows usage of git submodules.

8.30 How to debug page fetch

Debugging page fetch can be both easy and hard, depending on how much work you need to find the cause of the problem. You will find here some common and uncommon page fetching issues that happens on websites along it's fixes:

8.30.1 *no_url_encode: true*

This option forces a page to keep it's url as is, since DataHen decode and re-encode the url so it fix any error on it by default, useful to standardize the url for cache.

Example:

```
pages << {
  'url' => 'https://example.com/?my_sensitive_value'
}
# => url is re-encoded as "https://example.com/?my_sensitive_value="

pages << {
  'url' => 'https://example.com/?my_sensitive_value',
  'no_url_encode' => true
}
# => url is left as is "https://example.com/?my_sensitive_value"
```

8.30.2 *http2: true*

This change the standard fetch from HTTP/1 to HTTP/2, which not only makes fetch faster on websites that support it, but also helps to bypass some anti-scrape tech that usually blocks HTTP/1 requests.

Example:

```
pages << {
  'url' => 'https://example.com'
}
# => page fetching will use HTTP/1

pages << {
  'url' => 'https://example.com',
  'http2' => true
}
# => page fetching will use HTTP/2
```

8.30.3 response headers and request headers are different

There has been a few times on that a dev includes a response header within *headers: {}* causing the fetch to fail on websites that validates the headers it receives, so try to check which headers your browser shows on dev tools to see if an extra header is being used by mistake.

Example: let's say a page enqueues this way

```
pages << {'url' => '[https://www.example.com](https://www.example.com/)'}
```

then it will fetch, and then got response_headers like

```
response_headers: {
  'content-type' => 'json'
}
```

so on next page you enqueue the following page adding one or more response headers by mistake

```
pages << {
  'url' => 'https://www.example.com/abc'
  'headers' => {
    'content-type' => 'json'
  }
}
```

On this example, using *content-type* is fine on request as long as it is POST method, but this one is GET, so on this case this would be invalid and a website that validates the headers will fail.

8.30.4 bzip compression headers

Most browsers will include a headers indicating to compress the page to bzip or other compression format, most of the times it will not affect anything, but there are a few on which including these headers, will cause the content to fail.

8.31 Advanced Usage

8.31.1 Parsing Failed Responses

DataHen comes with a lot of safety harnesses to make scraping easy and delightful for developers. What this means is, we only allow for successfully fetched pages to be parsed. However, if you do need to go down into the detail and deal with your own failed pages, or other type of responses, we allow you to do so. On your config.yaml, add the following:

```
parse_failed_pages: true
```

After doing the above, don't forget to deploy your scraper, and restart your job.

We have now removed your safety harnesses. From now on, you have to deal with your own page reset, and page response statuses. Typically, you should have your parser deal with two kinds of responses, successful and failed ones. Look at the following example parser file on how we deal with the different responses in the same parser:

```
if page['response_status_code'] # if response is successful
   body = Nokogiri.HTML(content)
elsif page['failed_response_status_code'] # if response is not successful
   body = Nokogiri.HTML(failed_content)
end

doc = {
   text: body.text,
   url: page['url']
}

outputs << doc
```