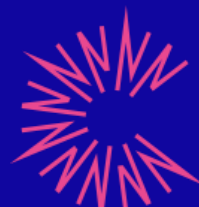


DAX Robot Primer

Overview and reference guide to DAX robot development

Version 0.1



ChannelSight®
Know more. Sell more.

1 Overview

1.1 Purpose

This document is intended as a brief primer on the core activities of the DAX team in maintaining and developing robots. The emphasis will be on the basic use case of Crawling and Updating which are the two primary robot types from which all other types are derived.

1.2 Version Information

Version	Author	Date	Notes
0.1 Draft	Alan O'Sullivan	2021-01-13	Draft
0.2 Draft Revision	Alan O'Sullivan	2021-01-15	Revised product codes, style guide
0.3 Revision	Alan O'Sullivan	2021-01-25	Add advanced topic for prepareRequests

Contents

1	Overview	1
1.1	Purpose	1
1.2	Version Information.....	1
2	Introduction	4
2.1	Overview	4
2.2	Example.....	4
2.3	Key Concepts	4
3	Development Guide	5
3.1	Overview	5
3.2	Apify Robot Types	5
3.2.1	Cheerio-Scraper https://apify.com/apify/cheerio-scraper	5
3.2.2	Puppeteer-Scraper https://apify.com/apify/puppeteer-scraper	5
3.3	Determining Robot Type	5
3.3.1	Overview.....	6
3.3.2	Site Examination	6
3.3.3	Optimize	8
3.4	Inputs	8
3.5	Output.....	9
3.5.1	Ratings.....	11
3.6	Crawlers	11
3.6.1	Overview.....	11
3.6.2	Types	11
3.6.3	Request Queue Management.....	11
3.6.4	Crawler Specific Output.....	14
3.7	Updaters.....	15
3.7.1	Overview.....	15
3.7.2	Stock.....	15
3.7.3	Price.....	16
3.7.4	Updater Specific Output.....	16

3.8	General Robot Configurations	17
3.8.1	Settings	17
3.8.2	Input Config	17
3.8.3	Advanced Configuration	18
3.8.4	Conventions	18
4	Code Style Guide	21
4.1	Overview	21
4.2	General Approach	21
4.3	Style	22
4.4	Error Handling	22
5	Advanced Topics	23
5.1	Inner requests	23
5.2	Additional Mime Types	23
5.2.1	Prepare Request Function	24
5.2.2	Development	24
5.2.3	Example: Homedepot.com	24
5.2.4	Input Manipulation	25
5.3	Cookies	25
5.4	Authentication	25
5.5	Cookies	26
5.6	Authentication	26
5.7	Blocks	26
6	Proxies	27
6.1	Overview	27
6.2	Proxy Types	27
6.3	Local Development / QA	27
7	Verification	28
8	Required Reading	29

2 Introduction

2.1 Overview

The core business use case is for the “Buy it Now” or “BiN” product. This allows our customers (brands) to integrate a widget on their site which directs customers to retailer product pages to purchase a product. It can also show a user the current price and stock for the products at those retailers. This information is updated daily by our robots.

2.2 Example

Refer to the following link for live video examples of how the product works: <https://www.channelsight.com/where-to-buy-platform-examples>

2.3 Key Concepts

The following table briefly explains the key concepts, terms and general nomenclature used by the robot platform and associated technologies:

Item	Description	Note
Actors	A microservice that can perform web automation. Takes an input configuration, is executable and saves results	A definition and can be used as a template for tasks
Tasks	A named instance of an actor with saved inputs. Are made to be repeatable running versions of Actors with varying inputs	An actor instance
Cheerio-Scraper	Version of an Actor that uses a simple http request mechanism and the cheerio framework to scrape a site	Cheerio
Puppeteer-Scraper	Version of an Actor that uses the puppeteer headless chrome browser to scrape a site	Puppeteer
Scraper	Denotes any automated mechanism that can extract data from a site.	
Crawler	Web scraper that <i>crawls</i> search pages to find products	Multiple search inputs can be configured
Updater	Scraper that visits a single web page to <i>update</i> product data points	Multiple pages can be sent to a robot for parallel processing
RequestQueue	The managed or self-managed list of URLs to process. Self-managed queues are generally required for crawlers.	See 3.6.3 Request Queue Management for more
Compute Unit	The cloud metric for measuring task usage and performance. Based on Actor/Task memory footprint and run time. This is the key to balancing performance, concurrency, and cost.	1024 MB of memory for 1 hour consumes 1 CU

3 Development Guide

3.1 Overview

This section illustrates the key development concepts and shows examples for the primary “Crawler” and “Updater” robot types.

3.2 Apify Robot Types

On the Apify platform, there are 2 pre-built web scrapers that are used in 99% of all cases. 95% of those use the Cheerio-Scraper type explained below.

This reduces the need to manage a lot of code as we only need to write code for the “*pageFunction*” which is where the data extraction occurs. All other aspects of the scraper are managed via simple configuration and settings directly in the UI

The cloud Actors code is available here with an explanation of each: <https://github.com/apify/actor-scraper>

3.2.1 Cheerio-Scraper <https://apify.com/apify/cheerio-scraper>

This scraper is built on NodeJS *request* SDK but enhances it to emulate a user browsing a web page by providing session and header information. This is an extremely fast scraper since it only loads a single request.

This uses the underlying framework of the *CheerioCrawler* but provides most of the properties and settings as simple configuration available on the UI as opposed to code.

To develop this robot type locally we must use the “*CheerioCrawler*” SDK and emulate the cloud application using code. This is illustrated in the provided templates.

3.2.2 Puppeteer-Scraper <https://apify.com/apify/puppeteer-scraper>

This scraper is built on puppeteer which provides a headless browser SDK to emulate a user using a real browser. It uses a lot of CPU and RAM and is costly to run and complicated to build.

This uses the underlying framework of the *PuppeteerCrawler* but provides most of the properties and settings as simple configuration available on the UI as opposed to code.

To develop this robot type locally we must use the “*PuppeteerCrawler*” SDK and emulate the cloud application using code. This is illustrated in the provided Puppeteer templates.

Note this robot is extremely inefficient thus should only be used in extreme cases where Cheerio-Scraper cannot be used. This can happen in circumstances where blocks are difficult to get past without emulating an actual browser.

3.3 Determining Robot Type

3.3.1 Overview

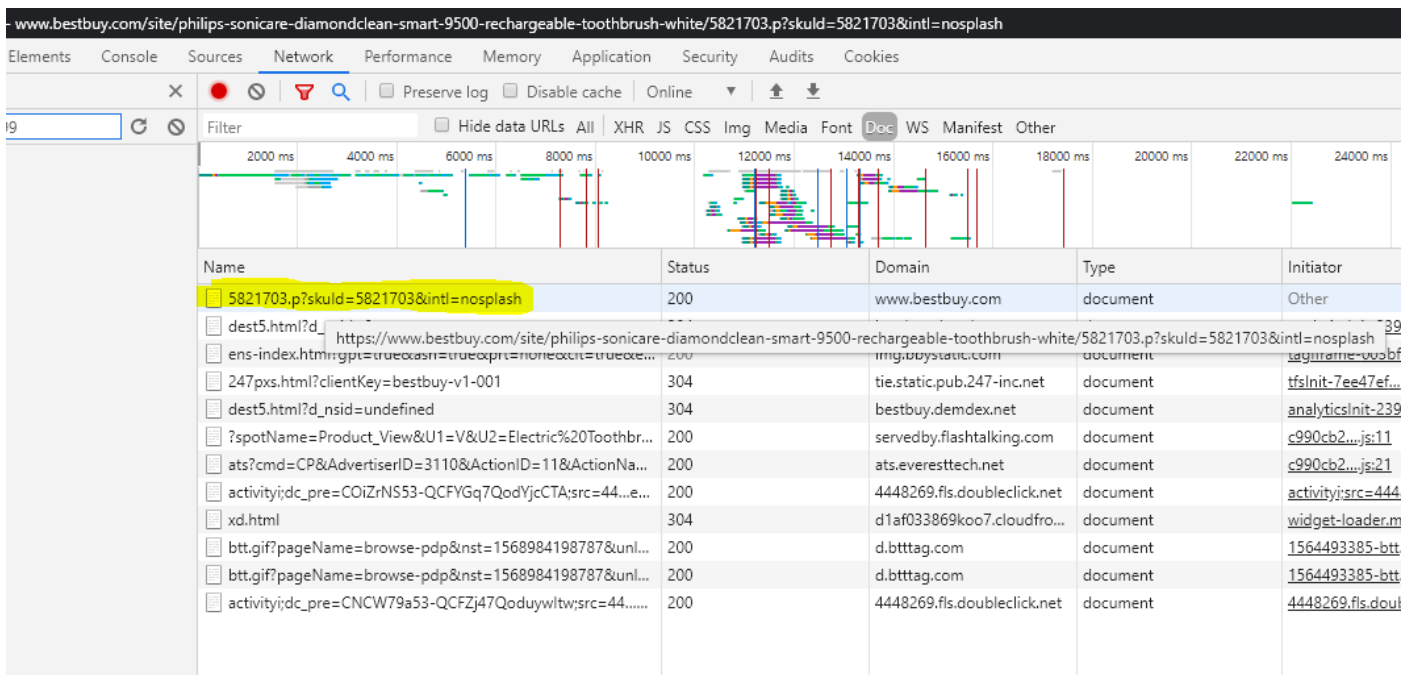
The most important aspect before building a robot is to determine the sites amenability to the most optimized type of crawl. This should be done in the following order:

1. Cheerio-Scraper – fastest and easiest to build
2. Puppeteer-Scraper – slowest to build and run – **only to be used as a last resort**

3.3.2 Site Examination

To ascertain the sites suitability for a particular type of scraper, the product page (Updater) and product listings page (Crawler) from the site must be examined fully. The following is an overview of the set of steps that should be followed for all sites.

- Open a chrome window and have the developer tools for the tab/window open on the “Network” tab (F12)
- Load the page to be examined by entering either the search URL or product page URL
- Once the page is loaded, first filter the tab by the “Doc” filter
 - This represents the individual server requests that are made to the server and the response of each of those requests
- Examine the “Doc” request. This will be the page that was requested that was entered into the browser URL. I.e., The HTML page that is requested.
 - Example:



The screenshot shows the Chrome DevTools Network tab with the 'Doc' filter applied. The first request in the list is highlighted:

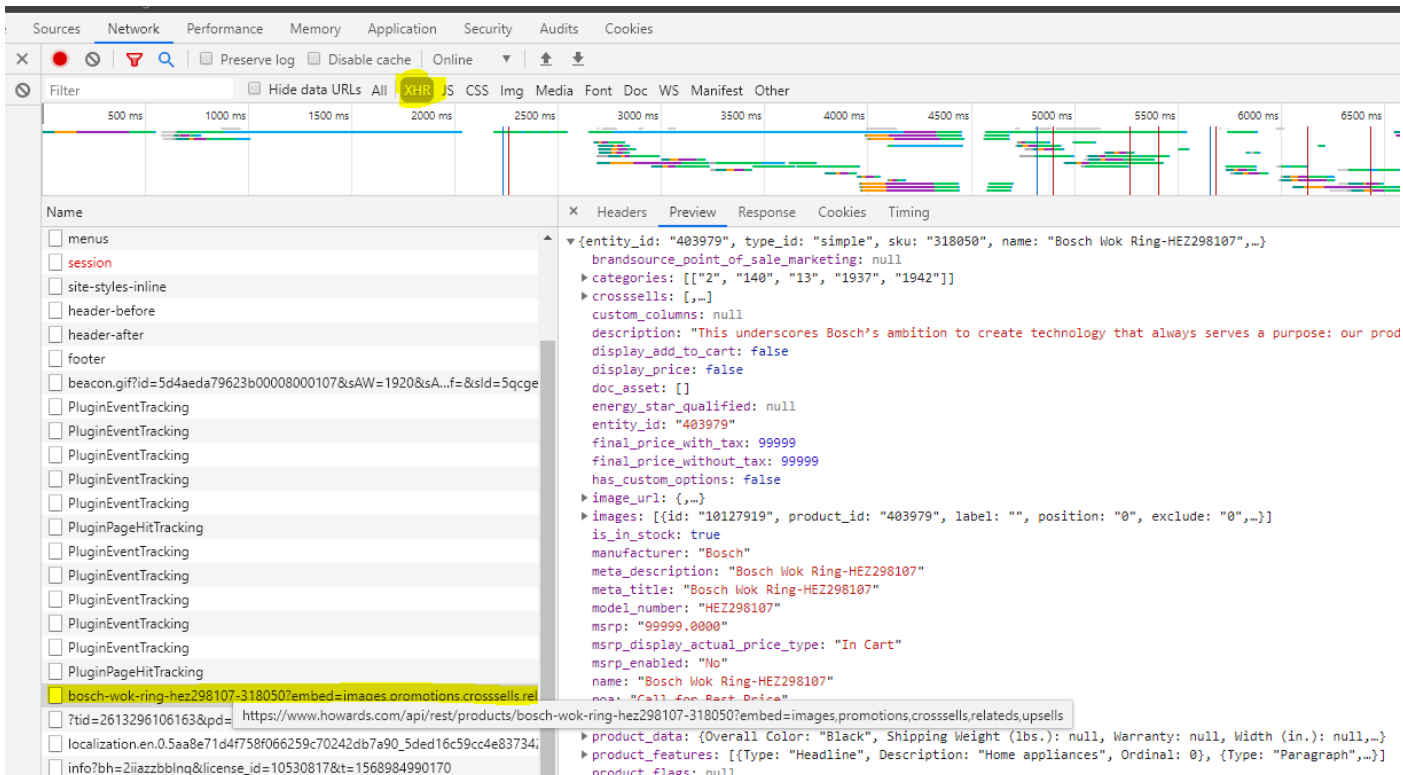
Name	Status	Domain	Type	Initiator
5821703.p?skuld=5821703&intl=nosplash	200	www.bestbuy.com	document	Other
dest5.html?d...	200	https://www.bestbuy.com/site/philips-sonicare-diamondclean-smart-9500-rechargeable-toothbrush-white/5821703.p?skuld=5821703&intl=nosplash	document	...
ens-index.html?gpt=true&asn=true&opt=none&ext=true&e...	200	img.bysstatic.com	document	...
247pxs.html?clientKey=bestbuy-v1-001	304	tie.static.pub.247-inc.net	document	...
dest5.html?d_nsid=undefined	304	bestbuy.demdex.net	document	...
?spotName=Product_View&U1=V&U2=Electric%20Toothbr...	200	servedby.flashtalking.com	document	...
ats?cmd=CP&AdvertiserID=3110&ActionID=11&ActionNa...	200	ats.everesttech.net	document	...
activityj;dc_pre=COiZrNS53-QCFYgQ7QodYjcCTA;src=44....e...	200	4448269.fl.doubleclick.net	document	...
xd.html	304	d1af033869koo7.cloudfro...	document	...
btt.gif?pageName=browse-pdp&nst=1568984198787&unl...	200	d.btttag.com	document	...
btt.gif?pageName=browse-pdp&nst=1568984198787&unl...	200	d.btttag.com	document	...
activityj;dc_pre=CNCW79a53-QCFZj47Qoduywltw;src=44.....	200	4448269.fl.doubleclick.net	document	...

- In this example, the document provided by the server is the URL that our request was redirected to
- Examine the contents of the document by selecting the relevant item (typically towards the top of the list and usually has some product identifier present)
- In the “Response” tab of the request, look for all relevant data points – price, stock etc... **This will inform you if the data is present in the HTML Doc request.**

- If all data points are present, this page can be scraped with the Cheerio-Scraper actor type using simple page request and CSS/HTML selector extraction code.

If the data is **not** contained within the document, it is likely that it was loaded via an XHR request. I.e. an Asynchronous JavaScript request (AJAX), typically found in pages that load only one page initially and dynamically change the content as a user navigates. An example is “load more” button where results are appended to existing listings rather than re-loading the entire page. Or “single page applications” where the URL never changes but the content of the page is dynamically loaded when a user interacts or requests new elements.

- Filter the Network tab by “XHR” tab – this displays dynamic requests made after the initial page load
- The format of these requests can vary greatly, but by traversing each one and using the “Preview” section of the view area, we can examine the content to see if the product/search data is contained within
 - The XHR URL’s usually contain the word “api” and/or the product code in the request but not always
- Sometimes, the data we are looking for is composed of multiple requests – general info, price and stock
 - In this case we can stitch the requests together in a robot to get all of the data we need
- Example:



The screenshot shows the Chrome DevTools Network tab with the 'XHR' filter applied. A list of network requests is shown on the left, with the selected request highlighted in yellow. The 'Preview' tab is active, displaying the JSON response of the selected XHR request. The response contains product details for a Bosch Wok Ring, including categories, description, price, and images.

```
{
  "entity_id": "403979",
  "type_id": "simple",
  "sku": "318050",
  "name": "Bosch Wok Ring-HEZ298107",
  "brandsource_point_of_sale_marketing": null,
  "categories": [
    "2",
    "140",
    "13",
    "1937",
    "1942"
  ],
  "crosssells": [
    {
      "entity_id": "403979",
      "final_price_with_tax": 99999,
      "final_price_without_tax": 99999,
      "has_custom_options": false,
      "image_url": {
        "id": "10127919",
        "product_id": "403979",
        "label": "",
        "position": "0",
        "exclude": "0"
      },
      "is_in_stock": true,
      "manufacturer": "Bosch",
      "meta_description": "Bosch Wok Ring-HEZ298107",
      "meta_title": "Bosch Wok Ring-HEZ298107",
      "model_number": "HEZ298107",
      "msrp": "99999.0000",
      "msrp_display_actual_price_type": "In Cart",
      "msrp_enabled": "No",
      "name": "Bosch Wok Ring-HEZ298107",
      "product_data": {
        "Overall Color": "Black",
        "Shipping Weight (lbs.)": null,
        "Warranty": null,
        "Width (in.)": null
      },
      "product_features": [
        {
          "Type": "Headline",
          "Description": "Home appliances",
          "Ordinal": 0
        },
        {
          "Type": "Paragraph"
        }
      ],
      "product_flags": null
    }
  ]
}
```

Some network requests made via XHR may not immediately be visible. One way to determine if a site supports smaller requests over an XHR API is to toggle filters on the page as you are examining the network requests XHR tab. Sometimes toggling a filter or going to the next page will reveal where the data is coming from as request are made on-demand to populate data.

The next step is to figure out how to use these requests:

- Inspect the “Headers” of the request to understand how the request was formed. The 2 most important header properties are:
 - **Method:** GET / POST
 - **Mode:** cors / no-cors (cross origin resource sharing) <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
- If the request was a “GET” method, right click the link and select “Open in new Tab”
 - If all of the data displayed in the “Preview” tab runs in a regular chrome tab, we can access this data directly
- If the request was a “POST” method, right click the request and select “Copy > Copy as Fetch” and execute the result in a JavaScript snippet from the “Sources” tab <https://developers.google.com/web/tools/chrome-devtools/javascript/snippets>
 - If all of the data can be retrieved, test the endpoint in an API test tool like Postman or the chrome add on <https://chrome.google.com/webstore/detail/restlet-client-rest-api-t/aejoelaoggembcahagimdiliamlcdmfm?hl=en>

3.3.3 Optimize

Whenever creating a scraper be creative and explore shortcuts. This can be done by analyzing the site structure in the chrome developer tools and using the console and snippets feature to query the DOM.

- Look for public APIs from a website via chrome inspection tools
- Look for embedded JSON which contains the complete product information in a single element
 - This can be serialized into a JSON object for easy transformation to our output format
- Use Meta properties in the HTML “head” node where available (typically for updaters)
 - E.g. <meta property="og:image" content=<https://assets.mmsrg.productImage.png>>
- Use data attribute selectors where possible to extract values rather than style/markup selectors which tend to change less often
- Look for form posting options which can be executed via code rather than simulated user inputs
 - When using puppeteer types this is particularly useful
- When JSON endpoints are secured and thus can only be executed in the browser context, reverse engineer the endpoint using chrome dev tools and re-create in the pageFunction method as a fetch (for puppeteer types)
 - When this can be done, the request blocking filters can be extremely aggressive since we don't need the rendered page

3.4 Inputs

The inputs for both “Crawlers” and “Updaters” will be supplied by the ChannelSight portal which will execute robots via the API and provide the inputs directly.

The inputs for Crawlers are stored as search strings in a configuration for each robot.

The inputs for Updaters are dynamic as a result of products found in a previous crawl that have been mapped to a brands catalog product.

1. Updater Example:

```
{
```

```
{  
  "url": "https://www.auchan.pt/422444/",  
  "userData": {  
    "Manufacturer": "Duracell"  
  }  
}
```

2. Crawler Example:

```
{  
  "url": "https://www.auchan.pt/Frontoffice/search/duracell",  
  "userData": {  
    "Manufacturer": "Duracell"  
  }  
}
```

The “Manufacturer” value will always be supplied on the incoming request in the “userData” object thus is accessed like so:

```
const manufacturer = request.userData.Manufacturer;
```

In general, it is possible to inject various properties and values into the *userData* object. From static values to identifiers and properties previously stored in our database related to the record being scraped on either a Crawl input or an Updater input.

3.5 Output

The output adheres to the strict schema that the portal will import.

Note that while this is a single interface, the “Crawler” and “Updater” will provide a different sub-set of this data as defined in each of those sections.

Property	Type	Required	Description	Note
ProductId	String	Mandatory	Unique Product Identifier. This is the retailer’s own unique product identifier	Crawler and Updater must use the same value
Manufacturer	String	Mandatory	The name of the Manufacturer/OEM.	Passed from the input
ProductName	String	Mandatory	The name of the Product	Crawler and Updater must use the same value

ProductUrl	String	Mandatory	The URL of the Product on the retailers site for an updater and the search URL for a crawler	Passed from the input for an updater
ProductCodes	Array	Optional (See: Note)	<p>An optional collection of additional product codes (EAN, UPC, GTIN, etc...) which is utilized by our retailer to manufacturer product matching system, in cases where the mandatory ProductId is an internal retailer product Identifier. See ProductCode</p> <p>Note: If the ProductId supplied is an internal retailer product Identifier, then you should endeavour to include an industry standard product identifier in this section</p>	Crawler only
Category	String	Optional	The category of the product (e.g. FMCG, Electronics)	
CurrencyCode	String	Optional	3 Character ISO Currency Code if Price is being provided	
Price	Double	Optional	Price is an optional value for crawlers but mandatory for updaters	
Stock	String	Optional	<p>Stock/Availability level. Supports only these values, in English:</p> <ul style="list-style-type: none"> a. InStock b. OutOfStock <p>If there is a discrepancy between a crawler listing page then exclude the stock value from the crawler</p>	
StockLevel	String	Optional	Supports strings such as: "Between 10-20 in stock", "50 units", "50 pcs", "10-20"	Updater only
Images	Array	Optional	An optional collection of image URLs for rendering a picture of the product on a brand manufacturer advertising site as it appears on the retailer. See Image	Crawler only
ImageUri	String	Optional	The single image of the page	Crawler only
LocalizedAddresses	Array	Optional	An optional collection of product URLs. The collection contains all product URLs for each culture info that appears on the retailer site. See LocalizedAddress	Crawler only

AdhocDataAttributes	Array	Optional	An optional collection of ad-hoc data attributes that are not part of the standard model and are product specific.	Updater only. Not required currently
---------------------	-------	----------	--	--------------------------------------

3.5.1 Ratings

Where ratings data is available for a product, include it in the output. Ensure a safety check is in place to prevent the robot from crashing if ratings information is not available. These data points can be from a Crawler or Updater.

Overall, these are optional depending on retailer support but if provided all properties should be included.

Name	Type	Description
RatingType	String	5-Star, 10-Star, Like/Dislike, Percentage
RatingSourceValue	Number	The actual value depending on the rating type
ReviewCount	Number	The count of user ratings/reviews on the product
ReviewLink	String	The specific product reviews link to the area of page with reviews. Typically, an internal “#” anchor. If no specific review link can be found this should default to the product page (request.url)

3.6 Crawlers

3.6.1 Overview

Crawler robots are the most important robots we have. This is because they are responsible for finding the brands products on retailers’ site. They are also the more difficult than standard updaters for 2 reasons:

1. They are required to find *all* brand products but also to make sure they **do not** find any off-brand products.
2. They are more complex as we must manage pagination (as a search input can product multiple pages) and potentially processing the actual product page to extract product codes.
 - a. This means manually dealing with the request queue (<https://sdk.apify.com/docs/api/request-queue>)

3.6.2 Types

There are 2 internal “types” of crawlers that vary by function:

1. Full Crawler (FC) – visits every page found in a search list to get product codes
2. Hybrid Crawler (HC) – only extracts data from the listings page so doesn’t visit product pages directly. **Note** this is the most efficient crawler and should be used where possible

3.6.3 Request Queue Management

When utilizing the request queue for adding pages or products on the fly when crawling, the code should reflect obvious mechanisms to ensure a simple, logical, and readable pattern. This is achieved by utilizing the “*userData*” object on the request to set properties and interpret those values in the *pageFunction* to handle accordingly.

When queuing new requests, we must also pass any data that originally came with that request. This is typically the “Manufacturer” property which is contained on the “*userData*” object but can include other properties as well.

3.6.3.1 Simple Pagination Queuing (Hybrid Crawler)

Where we queue pages from the initial search URL, we can add a “*queued*” Boolean to the requests to ensure this logic only applies to the first request.

Most pagination can be done upfront from the initial request if the number of pages returned can be calculated. This is more efficient than crawling via a “next” button on each request as we can avail of parallel processing if queued up front. There are multiple ways to find out how many pages there are.

- Total result count is displayed and the number of products per page is known (as in example below)
- Total number of pages is displayed on the site via the pagination control and the last page
 - <https://epicentrk.ua/ua/search/?q=bosc>
- Total number or products is contained beside each filter and thus is the total of each filter

In some extreme cases we need to queue pages individually using the “next” button mechanism on each page request until there are no more pages to process. This is the least optimal strategy

This sample is typically used for a “Hybrid Crawler” as defined in section [3.6.2 Types](#)

Note: This code is for illustrative purposes.

```
const handlePageFunction = async ({ response, request, $ }) => {

  //if the incoming request is not already queued then we are on the first request
  if (!request.userData.pageQueued) {

    const domain = "https://www.argos.co.uk/";
    const productsPerPage = 30;
    const searchResultsCount = $('div[data-search-results]').attr('data-search-results'); //120
    const pagesToQueue = Math.ceil(searchResultsCount / productsPerPage); //4
    const manufacturer = request.userData.Manufacturer;

    //queue pages
    for (var i = 1; i < pagesToQueue; i++) {

      //construct the next search page
      var nextPage = `${domain}search/${manufacturer}/brands:${manufacturer}/opt/page:${i + 1}`;
      log.info(`Queueing ${nextPage}`); //helpful logging
      var nextRequest = {
        url: nextPage,
        userData:
```

```
        {
            pageQueued: true, //flag as queued so we don't do this again
            Manufacturer: manufacturer //copy the manufacturer to the next request
        }
    }
    await enqueueRequest(nextRequest);
}
}
//scrape the products from the page listing...
```

3.6.3.2 Product Page Queuing (Full Crawler)

When queuing pagination and product pages, the pattern should be the same.

This sample is typically used for a “Full Crawler” as defined in section [3.6.2 Types](#)

Illustrative Code Flow inside the *pageFunction*

- Pagination – queue all found pages. See [3.6.3.1 Simple Pagination Queuing](#)
- Product Queue – queue all found product for all pages

```
//if not on a product page we are on a listing so queue each product for processing
if (!request.userData.productPage) {
    //selector for the products on the listing page
    const productCards = $('div[itemtype="http://schema.org/Product"]');
    //queue each product page for a full scrape of the page
    productCards.each(async function (index, element) {

        var pageUrl = $(element).attr('href');
        var pageRequest = {
            url: pageUrl,
            userData:
            {
                pageQueued: true, //flag as queued so it is not queued again
                productPage: true, //flag as as a product page
                Manufacturer: manufacturer //copy the manufacturer to the request
            }
        }
        await enqueueRequest(pageRequest);
    });
}
```

- Product Processing – extract data from the product page (example uses short-hand code for brevity)

```
//finally if a request is a product page then scrape the data to the output
if (request.userData.productPage) {
    const product = {
```

```

    ProductName: productNode.find('h1').text(),
    ImageUri: $("meta[property='og:image']").attr("content"),
    ProductId: productNode.find("td[itemprop='gtin8']").text(),
    Stock: productNode.find('.addToCartSubmit').text() == 'avail' ? 'InStock' : "OutOfStock",
    Price: productNode.find("span.price>em[itemprop='price']").attr('content'),
    ProductUrl: request.url, //copy from input
    Manufacturer: request.userData.Manufacturer, //copy from input
    ProductCodes: [
      {
        ProductCodeType: "EAN",
        Value: productNode.find("span.ean[itemprop='ean']").attr('content')
      }
    ]
  }
}
//save the product (use return on the cloud app)
await Apify.pushData(product);
}

```

3.6.4 Crawler Specific Output

While the output for all robots is a standard set of mandatory/optional values, a crawler and updater differ in what is important. See section [3.5 Output](#) for the entire output schema details.

Property	Type	Required	Description
ProductId	String	Mandatory	Unique Product Identifier. Should be the same for both updater and crawler
Manufacturer	String	Mandatory	The name of the Manufacturer. Passed to the input
ProductName	String	Mandatory	The name of the Product
ProductUrl	String	Mandatory	The URL of the Product on the retailer's site
ProductCodes	Array	Optional	If they can be found and match our catalogue. Note: For brevity we can use short-hand codes to simplify output
Price	Double	Optional	
Stock	String	Optional	Stock is an optional value for crawlers if it cannot be accurately ascertained
Images	Array	Optional	
ImageUri	String	Optional	The single image for the product. Note this is preferable to an images array for crawlers

LocalizedAddresses	Array	Optional	
--------------------	-------	----------	--

3.6.4.1 Short-hand Codes

To provide product codes in the output in a simpler way, we can use the following properties:

UPC

- EANCode
- CTINCode
- GTINCode
- ASINCode
- OTHERCode

Example:

```
var product = {
  ProductId: id,
  Manufacturer: request.userData.Manufacturer,
  ProductName: name,
  ProductUrl: request.url,
  Price: price,
  Stock: stockValue,
  EANCode: eanValue,
  ImageUri: imageValue
}
```

It is not necessary to use the correct type with what is on the retailer site as the retailer site itself might be wrong. We can plug any codes we find into any of the provided properties as they will be checked against all values in our product catalogue provided by the brand.

3.7 Updaters

3.7.1 Overview

In general, updaters are very straightforward to create. Since they are executed on a single web page, there is less complexity in comparison to crawlers. When pages can be scraped using the Cheerio-Scraper, the creation of this type of robot is extremely simple.

Note: A template is provided with the boilerplate code need to quickly setup a new updater robot

3.7.2 Stock

In general stock mapping options follow the schema.org <https://schema.org/ItemAvailability> enumeration as more and more retailer sites use structured data to mark-up their product pages.

The stock rules are usually provided in advance but should follow a single general rule:

This means if the user can add the item to the cart, it is usually considered in stock. There are several markers to conclude this is the case and vary from site to site.

3.7.3 Price

Price extraction should be formatted per the regions locale and rules requested in the task. All symbols and additional meta should not be included.

3.7.4 Updater Specific Output

While the output for all robots is a standard set of mandatory/optional values, a crawler and updater differ in what is important. See section [3.5 Output](#) for the entire output schema details.

Property	Type	Required	Description
ProductId	String	Mandatory	Unique Product Identifier. Should be the same as the crawler supplies originally
Manufacturer	String	Mandatory	The name of the Manufacturer. Passed to the input
ProductName	String	Mandatory	The name of the Product
ProductUrl	String	Mandatory	Copied from the input as previously crawled
Price	Double	Mandatory	
Stock	String	Mandatory	Stock is an optional value for crawlers if it cannot be accurately ascertained
Ratings	Object	Optional	Should be provided for all updater robots if available. See section 3.5.1 Ratings for more information

See the provided templates for examples.

3.8 General Robot Configurations

The following table illustrates the general configurations currently in use for each robot type. These are available as settings/input on the Apify cloud task and as code configurations in the locally developed equivalents.

3.8.1 Settings

Property	Default	Actor Type	Comment
Build	0.1.23	Cheerio-Scraper	Latest stable version
Build	0.1.28	Puppeteer-Scraper	Latest stable version
Timeout	7200	Both	2 hours but varies depending on estimated number of products.
Memory	256	Cheerio-Scraper	Simple requests require only a small amount of memory
Memory	2046	Puppeteer-Scraper	Since a browser is being emulated, requires more memory

3.8.2 Input Config

The primary inputs into each robot type are the *pageFunction* and the *prepareRequest (preGoTo)* function. These are specific per robot. However, there are a common set of configurations available to each robot:

Property	Default	Actor Type	Comment
StartUrls	Per retailer	Both	Sent by the ChannelSight portal from the database
PsuedoUrls		Both	Not used
LinkSelector		Both	Not used
KeepURLFragments	True	Both	Allows “#” values on URLs to be considered unique. Example: www.site.com/product-one www.site.com/product-one#variant-A
PageFunction	Per retailer	Both	Executed when the page is visited, and the HTML/JSON (response content) is provided inside for data extraction. This is the brains of the robot
ProxyConfiguration	Auto	Both	Proxy to funnel requests through to avoid detection
SessionPoolName			Not used

InitialCookies	Per retailer		If required to get access to a site. Note: Dynamic values like session IDs should never be used
PrepareRequest Function / PreGoTo Function	Per retailer	Both	Intercept the request before it executed to make any changes required. Example, changing the destination URL to an API endpoint for a product page
AdditionalMimeTypes	Per retailer	Cheerio	Allows us to request http endpoints that are XML/JSON
SuggestResponseEncoding			Not used
ForceResponseEncoding			Not used
IgnoreSSLErrors	True		We don't care about SSL

3.8.3 Advanced Configuration

Property	Default	Actor Type	Comment
MaxRequestRetries	2	Both	Defines how many retries we will make. Some problematic retailers can need 5 or more
MaxPagesPerRun			Not used
MaxResultsRecords			Not used
MaxCrawlingDepth			Not used
MaxConcurrency	5	Both	Defines max parallel requests we want to achieve. Can speed up processing but may need more memory to run and cause blocks
PageLoadTimeout	120	Both	Give sufficient time for the page to load
Logging	True	Both	Enabled on non-production environments. Requires correct use of the provided <i>log</i> class
CustomData	Per retailer	Both	Not used

3.8.4 Conventions

There is a limited user interface for management, classification, and general organization of crawlers. This means it is important to follow strict conventions for names of Tasks. This is essential to keep the environment clean and not polluted with test and/or redundant crawlers.

The listing pages on the platform are extremely basic. Pagination is via infinite scroll and there is a simple search filter to find robots by their “Name” field. Since the search is limited to the “Name” field, task names should follow the following **strict** conventions to ensure optimal search and logical organization within a limited feature:

{country_code}-{retailer}-{type}-{manufacturer(optional)}

The “type” field represents our internal representation of type:

Type	Description
HC	Hybrid Crawlers. Only extracts data from product page listings
FC	Full Crawler. Visits each product page found in the search results
PPU	Product Page Updater
PPUR	Product Page Updater with Ratings
Reviews	Reviews extraction robot
RTL	Real Time Lookup robot for localized price and stock
API	An updater or crawler that uses an authenticated API
PPUR-CC	Product Page Updater with Ratings and Content Compliance (images, videos, breadcrumbs)

Examples:

Actor	Type	Name
Cheerio	Crawler	UK-Argos-HC
Cheerio	Updater	US-Lowes-PPU--Saf
Cheerio	RTL	US-GNC-RTL
Puppeteer	Crawler	UK-Argos-HC
Puppeteer	Updater	US-Houzz-PPU
Puppeteer	RTL	US-AmazonFresh-RTL
API	Crawler	CH-Microspot-HC-API
API	Updater	CH-Microspot-PPU-API
API	RTL	US-Target-RTL-API

Note: No other information should be appended or prefixed to task names. To add contextual information, use the “Description” field in the task settings pages.

There is a 30-character limit to names so use common sense when required. E.g. Shorten the retailer site name if required and put the actual detail in the description.

4 Code Style Guide

4.1 Overview

Most complexities of crawling are handled by the Apify platform (proxy rotation, 403 handling, concurrency, masking etc.) and are hidden from the developer. This does not remove responsibility from the developer in creating clean clear code and handling errors or expected deviations appropriately.

4.2 General Approach

Since the code is minimal and the level of support may vary with little experience in development, the code must be written as simply as possible. It should also follow the general order of events that happen when a page is scraped:

1. Constant variable assignments
2. Handle 404
3. Paginate
4. Extract
5. Output result

Most Important concepts:

- Simplify code
 - Don't use anonymous functions
 - Don't use inline functions
 - Don't chain methods unnecessarily
 - Don't chain selectors and regex
 - Don't use "map" or "reduce" (filter can be used)
- For and if loop logic should be appropriate
 - There should be no code dead ends. Include logging if there is no viable alternative path
- Null exceptions are handled/logged
 - Example: if an element is not present it may indicate a 404 or 403 page and the input should be handled appropriately
- Error handling is appropriate (try / catch)
 - Example: If an anticipated error it should be handled/logged
- No hard coded access tokens/session variables etc... (in cookies especially)
- "Return" statements should only be used for returning output
- Use the "userData" object to control logic flow (see examples)
 - Don't use "switch" statements to control flow
- Correct use of *userData* and inputs to control code flow (avoid URL manipulation)
- Logging is appropriate for the level of complexity (and using "log" not "console")
 - log.debug() log.info() and log.error() usage
- No bespoke logic for specific Manufacturers. Rules should be generic and governed by input data on the *userData* object
 - Example: Seller rules in *userData*
- Add code comments where appropriate (if the code is not immediately clear or there is an obscure business rule)

4.3 Style

Use the following style rules:

- K&R indentation

```
if (x == y) {
    foo();
    bar();
}
```

- a. Put the opening bracket at the end of the first line
- b. Put the closing bracket on a new line without leading spaces
- c. Do not end a complex statement with a semicolon

- CamelCase naming convention
- Always put spaces around operators
- Always end a simple statement with a semicolon
- Use string interpolation for log statements by using a backtick and variables withing the string. Example:
log.debug(`Received \${response.statusCode} for \${request.url}`);

4.4 Error Handling

The following table captures some of the common use cases for error handling

Item	Type	Action
404	Page not found	Do not include the result in the output. Return an object to the output showing the detail of the 404. This ensures the product is not imported in the database. See template example.
403	Access denied	Should be checked only if using 0.1.17 of cheerio scraper. This is automatically failed using 0.1.23 and thus will be retried per the configured retry count in the input
Element not found	Extraction failed	If there is reason to believe in some cases an element won't be available, check for its existence before assuming extraction will succeed. Log an error if the element is not found
Regex Failed	Extraction failed	Handle complicated regex where we extract codes from product names etc.... for specific brands using a check for the result before trying to extract a match.
JSON Parse	Extraction failed	Use try/catch when parsing embedded JSON as there can be invalid characters present in some cases

In general, errors should not be swallowed in a try-catch or ignored. At the very least we should log errors to be able to debug on the cloud platform what went wrong.

5 Advanced Topics

5.1 Inner requests

In some cases, it may be necessary to call a public API from the page to get another value. This is prevalent for ratings data where it is often hosted by 3rd party data providers. Apify provides a request method that emulates a browser and can be used *inside* the *pageFunction* to make a request for the current page being processed.

See: <https://sdk.apify.com/docs/api/utills#utillsrequestasbrowseroptions> for detailed usage

Example: This excerpt from a *pageFunction* shows how a robot while scraping a product page uses an external API lookup to get the ratings for the product.

```
var keyInfo = JSON.parse($('id="__NEXT_DATA__").text()); //get the API key from the HTML
var reviewsApiKey = keyInfo.runtimeConfig.reviewsApiKey; //extract key
var schema = productNode.find('[type="application/ld+json"]').toArray().filter(x => JSON.parse(x).html().replace(/s+/g, " ")[ '@type' ] == "product");
var infoNode = JSON.parse(schema.html().replace(/s+/g, " "));
var sku = infoNode.productId; //query the ratings API by using the key and SKU
var productApi = `https://api.johnlewis.com/ratings-reviews/v1/reviews?productId=${sku}&pageNumber=1&pageSize=15&sort=useful&api_key=${reviewsApiKey}`;
var productJson = await Apify.uts.requestAsBrowser(
  {
    url: productApi,
    abortFunction: null,
    //ensure a proxy is used internally which should be the same as the configured proxy on the robot
    proxyUrl: "http://auto:FFFR2R3G0H4Fw9pWzVLSdzyj@proxy.apify.com:8000"
  });
var productData = JSON.parse(productJson.body);

//set ratings value on the product from the API
if (parseFloat(productData.overallRatings.value).toFixed(1) != 0.0) {
  product.ReviewCount = productData.summary.totalResults;
  product.RatingSourceValue = parseFloat(productData.overallRatings.value).toFixed(1);
  product.RatingType = "5-star"; //see supported types from version 2.7 of the RDP document
  product.ReviewLink = product.ProductUrl + '#ratings-and-reviews'; //e.g #customer-reviews
}
```

5.2 Additional Mime Types

To use any non-HTML resource, the “additionalMimeTypes” array must be populated with the expected content type so the correct headers are used internally by the underlying request.

This is set on the constructor of the local “CheerioCrawler” as “additionalMimeTypes” array. On the cloud app “Cheerio-Scraper” this property is exposed on the input.

Example:

```
const crawler = new Apify.CheerioCrawler({
  requestList,
```



```
    handlePageFunction,  
    prepareRequestFunction,  
    handleFailedRequestFunction,  
    proxyConfiguration,  
    ignoreSslErrors: true,  
    additionalMimeTypes: ["application/json"]  
  });
```

5.2.1 Prepare Request Function

In some cases, the product page URLs we save in our database and which a user will load in a browser, are not the definitive location of the data. Some pages are loaded via an API in the background. Apify Cheerio-Scraper provides an intercept function where we can change the location of the URL. We can change the request type and add content to the body in the case of "POST" type requests.

Crawler type robots generally don't need to use this intercept because the inputs are crafted by the developer. This means API URLs can be used directly as inputs. If other data is required to be added (POST body or additional headers) we may still need to use the prepare request function to enable more complex HTTP calls.

5.2.2 Development

In the local development using "CheerioCrawler" this method is called "prepareRequestFunction"

In the cloud actor using "Cheerio-Scraper" this method is exposed in the input as "preGotoFunction"

Examples are provided in both the templates and on the Apify environment.

5.2.3 Example: Homedepot.com

This site has an API for search which is a GET method:

A product page is located at: <https://www.homedepot.com/p/314024432> but the API to this page is located at: <https://www.homedepot.com/p/svcs/frontEndModel/314024432>

1. prepareRequest function:

```
const prepareRequestFunction = async ({ request, $ }) => {  
  
  //check if the "OriginalUrl" property was added  
  //if not it is a new request otherwise it is a retry  
  if (!request.userData.OriginalUrl) {  
    request.userData.OriginalUrl = request.url; //save the original input URL to use in the  
    var productId = request.url.split('/').pop();  
    request.url = `https://www.homedepot.com/p/svcs/frontEndModel/${productId}`; //transform the url  
  }  
}
```

2. pageLoadFunction

The “pageLoadFunction” can now access an object via the context called “json” which will contain the json data retrieved from the request. This can be parsed, and the relevant values extracted and added to the output as per any robot.

Local: `const handlePageFunction = async ({ response, request, json }) => { ...`

Cloud: `const { request, json, log } = context;`

5.2.4 Input Manipulation

5.2.4.1 Crawlers

When integrating with the ChannelSight portal, it is straightforward to add an API URL directly to a crawler as the inputs are manually configured. Therefore, there is no need to manipulate a URL using the aforementioned “prepareRequest” function so long as no additional headers or body manipulation is needed for a crawler.

5.2.4.2 Updaters

Again, if no headers or body manipulation is needed, Updater inputs can be manipulated before being sent to the robot using templates. From the example above, the robot could be written *without* a “prepareRequest” function manipulating the URL since we can tell our system to transform these URLs based on properties we already have saved for the pages (from the crawler).

Example: HomeDepot.com

The “input” configuration allows us to transform *all* product URLs with a simple mechanism.

`https://www.homedepot.com/p/svcs/frontEndModel/{{RPI.RetailerProductCode}}`

Where RPI.RetailerProductCode is the column from our RPI (RetailerProductInfo) table containing the product code. This allows us to avoid intercepting and changing requests dynamically.

Note: In these cases, it is critical to maintain the product URL as was originally saved. To do this we also pass in to the userData the originally saved URL. E.g. in this case of the format <https://www.homedepot.com/p/314024432> to make sure we don’t change the URL. This can also be done when altering the URL in the request by saving the URL in a different property before changing the request URL being made.

5.3 Cookies

In rare cases, a cookie will be required to make requests to pages with a session ID or a cloudflare ID that let’s the website know a user has visited the page with a regular browser. Since puppeteer can do this for us, we can use that to get the cookies.

Talk to a DAX developer for examples of how this is achieved as it greatly optimizes the robot run time.

5.4 Authentication

In rare cases, a website will require authentication to get access to the resources. There are many different types of authentication, but a basic type can be reverse engineered to build an authentication request to get required headers to make subsequent requests.

Talk to a DAX developer for examples on how this is achieved.

5.5 Cookies

In rare cases, a cookie will be required to make requests to pages with a session ID or a cloudflare ID that let's the website know a user has visited the page with a regular browser. Since puppeteer can do this for us, we can use that to get the cookies.

Talk to a DAX developer for examples of how this is achieved as it greatly optimizes the robot run time.

5.6 Authentication

In rare cases, a website will require authentication to get access to the resources. There are many different types of authentication, but a basic type can be reverse engineered to build an authentication request to get required headers to make subsequent requests.

Talk to a DAX developer for examples on how this is achieved.

5.7 Blocks

In some cases, a website will not load using either of the default Actors provided. This is usually a combination of proxy issues or by not using a specific version of a library. It can also be a result of cookies. If it seems impossible to scrape, contact a DAX developer for assistance.

6 Proxies

6.1 Overview

We use extensive proxy networks to mask our robots browsing to reduce block occurrences. Masking is provided by each robot type, but proxies should be used for every robot in local, QA and Production.

6.2 Proxy Types

There are several proxy types at our disposal, but we should aim to use the appropriate proxy for the appropriate use case. It can seem easy to use the “best” proxy all the time but that would always be the most expensive option.

Type	Description
Data Center (Apify)	Cheapest and fastest but can get blocked. Apify provides many of these and most of our robots are configured with “Auto” proxy handled by Apify
Data Center (Oxylabs)	50 US proxies which are generally faster than Apify so are used in exceptional circumstances where a combination of Apify provided ones don't work
Residential (Apify)	For stubborn sites where blocks are frequent with captchas, cloudflare and other bot detection mechanisms. Expensive solution so a calculation for bandwidth usage is required before using
Residential (OxyLabs)	NGRP proxies which are the most expensive but most capable proxies we have in our arsenal. https://docs.oxylabs.io/next-gen-residential-proxies/

6.3 Local Development / QA

Using your local environment, you will not encounter many blocks as your IP is a residential one by default. To simulate what will happen when the robot is deployed to QA and Production you must ensure every run locally is done with proxy configuration. This is provided in the Crawler and Updater templates.

7 Verification

Once a developer has completed the development of a robot (crawler or updater) the following process must apply:

1. Code review by a peer following the approach outlined in section [4 Code Style Guide](#)
2. Ensure general approach is as specified
 - a. Variable naming
 - b. Logic / flow control
 - c. Output schema conformance
 - d. Low complexity / High readability and maintainability
3. The crawl inputs are sufficient to find all the brands products
4. The robot has been executed for the relevant inputs.
 - a. For a crawler this means all search inputs are tested with the count of results from a manual search
 - b. For an updater this means all crawled page URLs are tested as inputs
5. The output has been verified against the required output for each robot.
 - a. Download and examine the output for any discrepancies in each data point. This can be a high-level sanity check for formatting / missing data

Any deviations from these points must be resolved before presenting the robot as complete.

8 Required Reading

The following links are mandatory reading to understand the Apify platform and SDK

Link	Description
https://sdk.apify.com/docs/guides/getting-started	Getting started guide. Ignore installation instructions
https://apify.com/apify/cheerio-scraper	Primary cloud actor we use
https://docs.apify.com/tutorials/apify-scrappers/cheerio-scraper	Cheerio tutorial
https://docs.apify.com/tutorials/apify-scrappers/puppeteer-scraper	Puppeteer tutorial