

Product Cataloging and Intelligence

...

(Team 17)

Saksham Aggarwal

Aditya Gaykar

Anil Kumar

Overview

Submission date

- 14 April, 2016

Recent progress

- Scope document
- Solution Approach document
- Phase 1 Submission : Basic scrapper and query interface

Current Submission

- Generic Scrapper interface, with reppy and dry scrape support
 - Web app development in Node.js server and MongoDB database
-

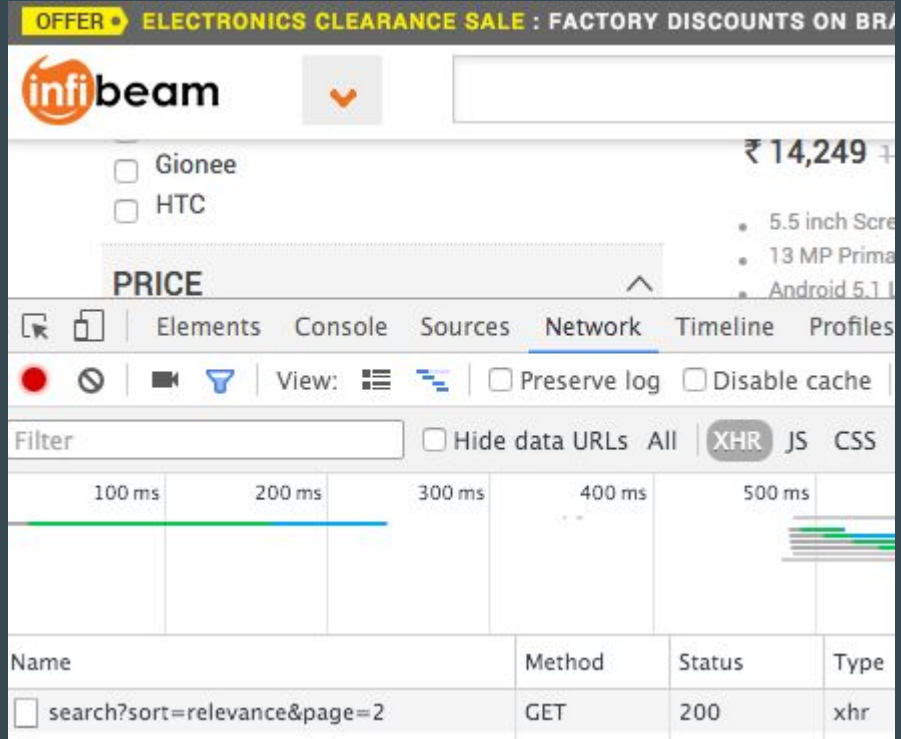
Target product website

- We have chosen www.infibeam.com as our target product website



Target product website structure

- We have focussed on mobile products on the url : www.infibeam.com/mobiles
- This page uses ajax calls to get more mobile phone entries, on scroll down event.
- On inspecting the network XHR requests in chrome we could track the pagination url:
 - www.infibeam.com/Mobiles/search?sort=relevance&page=<p-no>



Scraping Algorithm : Generate Seed URLs

1. `Generate_Seed_URLs(seed):`
 - a. `Queue Q = [] #for seed urls`
 - b. `Push initial seed url to Q`
 - c. `For page in range(1,n):`
 - i. `For mobile in page:`
 1. `Add mobile.url to Q`

Scraping Algorithm : Generate Master Dump

- `Generate_master_dump(Q):`
 - For url in Q:
 - `P =get(url).parse()`
 - `Product.data = P.product.data`
 - `Price.data = P.price.data`
 - `Vendor.data = P.vendor.data`
 - `dump_to_file(product,price,vendor)`

Implementation: Generalized Spider

- An Abstract class in Python (`base_scraper`). The workflow is as follows:
 - Load catalogue pages from target website one by one.
 - Extract all product URLs from catalogue page.
 - Scrape all products one by one, if not already scraped.
- For every site specific spider/scrapper inherit from `base_scraper` and override 3 functions necessarily.
- The above class was inherited to create a specialized scraper for our target site (`infibeam`)



Key Challenges

Scraper

1. Scrape sites that do not allow bots.
 2. Some sites are highly dynamic.
 3. Avoid overloading the site with requests.
 4. Respect the site's privacy and avoid scraping pages it doesn't want us to.
 5. Scraping is a slow process.
-

Scraper: Overcoming Challenges

- Scraping sites that don't allow bots:
 - The key idea was to mimic a browser. Sites don't block browsers from accessing pages.
 - Initially mimicked the Firefox browser by copy pasting its header string.
 - Later moved to a headless browser that handles this automatically
- Handling Dynamic Content:
 - Problem: Some sites(Infibeam in our case) change the product name and image dynamically using JavaScript based on the selected variant.
 - Problem with naive solution: Just getting the HTML page will yield the wrong details.
 - Solution: Use a headless browser like **dryscraper** which loads the complete webpage and runs the javascript as well.

Scraper: Overcoming Challenges

- Don't overload the website:
 - Some sites put a restriction on how frequently bots can access pages from them, known as the crawl delay.
 - Too many requests can hinder with the functioning of the website and we may get blocked!
 - Website administrators mention the Crawl-delay in their **robots.txt** file.
 - Before scraping the target website, we read the **robots.txt** and make sure there is at least 'Crawl-delay' difference between any 2 fetches.
- Respect site's privacy:
 - A site may not want you to scrape some webpages and scraping them may block you!
 - As a result, before we scrape any page, we check if we are indeed allowed to scrape it. This information is also present in the robots.txt file.
 - We've used **reppy** as for parsing the robots.txt file.

Scraper: Overcoming Challenges

- Speed:
 - Scraping is usually a slow process, and hence scraping the entire catalogue again and again is not feasible.
 - As a result we've broken the task into two subtasks
 - Scrape new Products only:

We look for new products on the target website and add them to our dump. Since old products are not scraped, only a small section of the products are downloaded.
 - Update a single product:

Since most products don't change frequently, we can request to update the information of a single product. This is matter of a couple of seconds, and we need not update products not viewed by the consumer.
 - The two subtasks, when combined give us at least as much power as the single task would've given us, and also maintains speed.

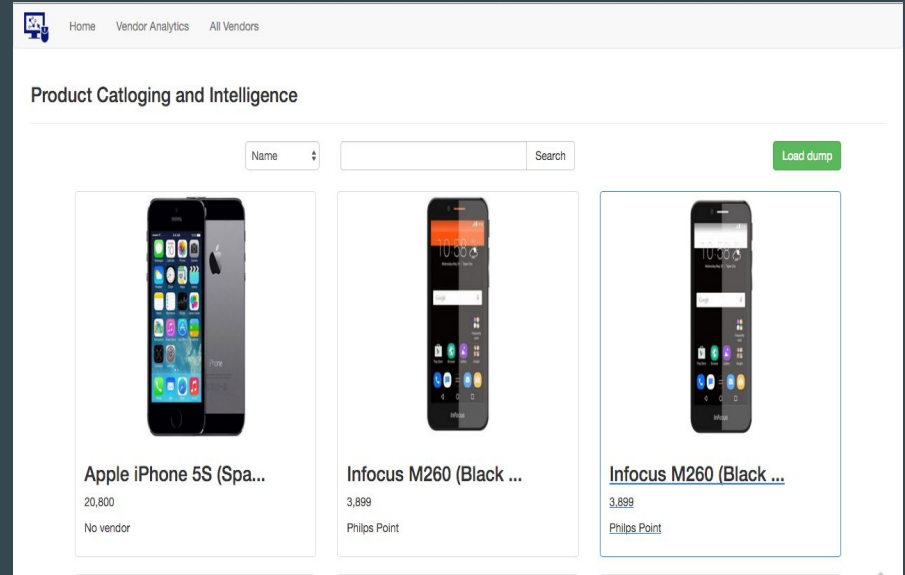
Implementation : MongoDB core master dump schema

- We choose mongodb due to its ability to handle unstructured data easily
- “price” and “updated_on” fields were maintained as object arrays
- So on every Resync call the new price values and updated_on time stamp would be added.
- This data organisation helped to further run complex queries in order to perform analytics on the scrapper data

```
3 // Declare schema
4 ▼ var mobileSchema = new mongoose.Schema({
5     name: {type: String, required: true},
6     url: {type: String, required: true},
7     vendor: {type: String},
8     price: {type: Array},
9     image: {type: String},
10    description: {type:String},
11    specs : {type:String},
12    updated_on: {type: Array},
13    created_on: {type: Date, default: Date.now}
14 });
```

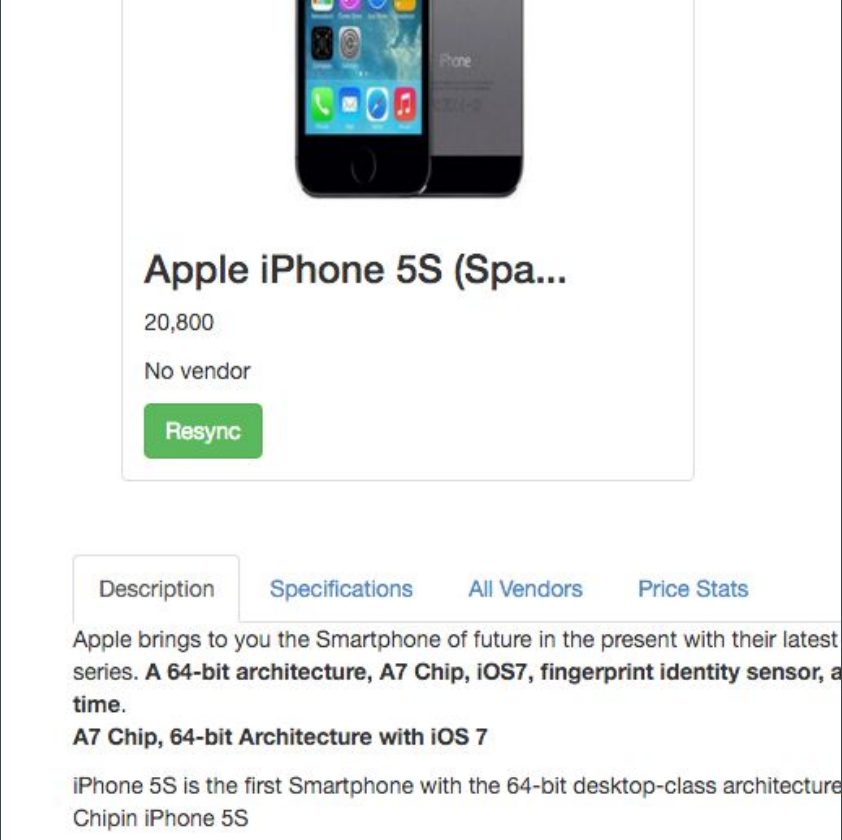
Implementation : Web application

- Web application is implemented with following technologies
 - Node.js server
 - MongoDB server
 - Mongoose MongoDB client for node
 - Twitter Bootstrap UI framework
 - Express MVC framework
 - Jade template engine
 - Google Charts API
 - jQuery
- Web application enables the user to navigate, query scrapped data in a well defined user interface
- Interfaces include, search interface to search products by name, vendor and specification
- Search feature is implemented using jQuery ajax calls.



Web Application : Product View & Resync Feature

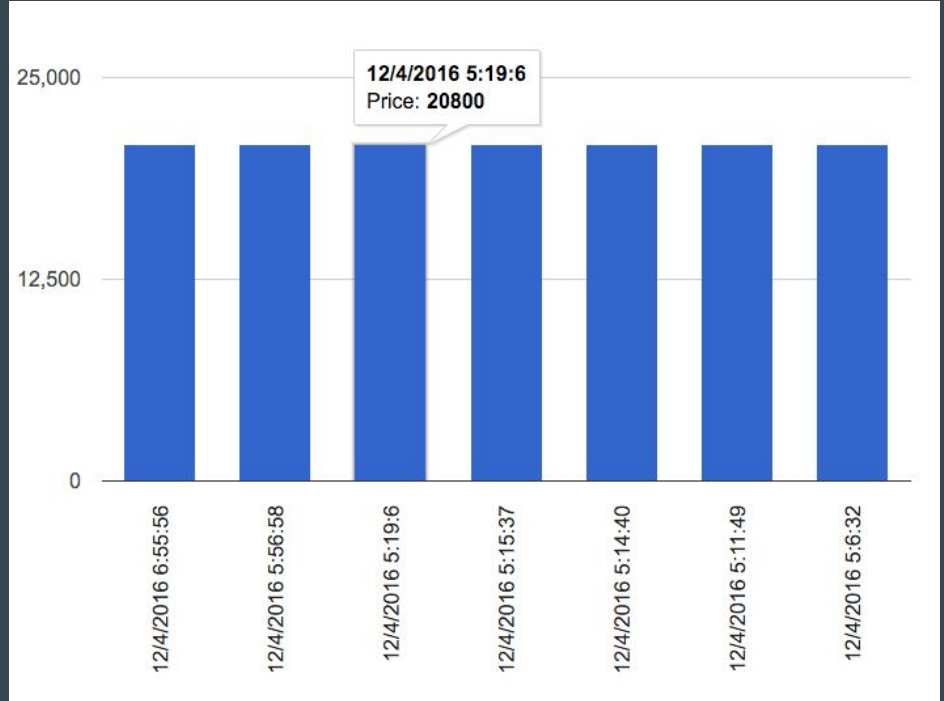
- Product view interface to see more information about a particular product
 - Description
 - Specifications
 - All Vendors
 - Price stats
- Resync feature is an important feature to have for the user, which when clicked initiates a dry scrape script to fetch in the latest information for that particular product
- This information is logged into the mongodb database can be verified in Price stats section



The screenshot displays a product view interface for an Apple iPhone 5S. At the top, there is an image of the iPhone 5S. Below the image, the product title "Apple iPhone 5S (Spa..." is shown, followed by the price "20,800" and the text "No vendor". A green button labeled "Resync" is positioned below the price. Below the product information, there are four tabs: "Description", "Specifications", "All Vendors", and "Price Stats". The "Description" tab is currently selected, showing the following text: "Apple brings to you the Smartphone of future in the present with their latest series. A 64-bit architecture, A7 Chip, iOS7, fingerprint identity sensor, a time. A7 Chip, 64-bit Architecture with iOS 7". Below the description, it states "iPhone 5S is the first Smartphone with the 64-bit desktop-class architecture Chipin iPhone 5S".

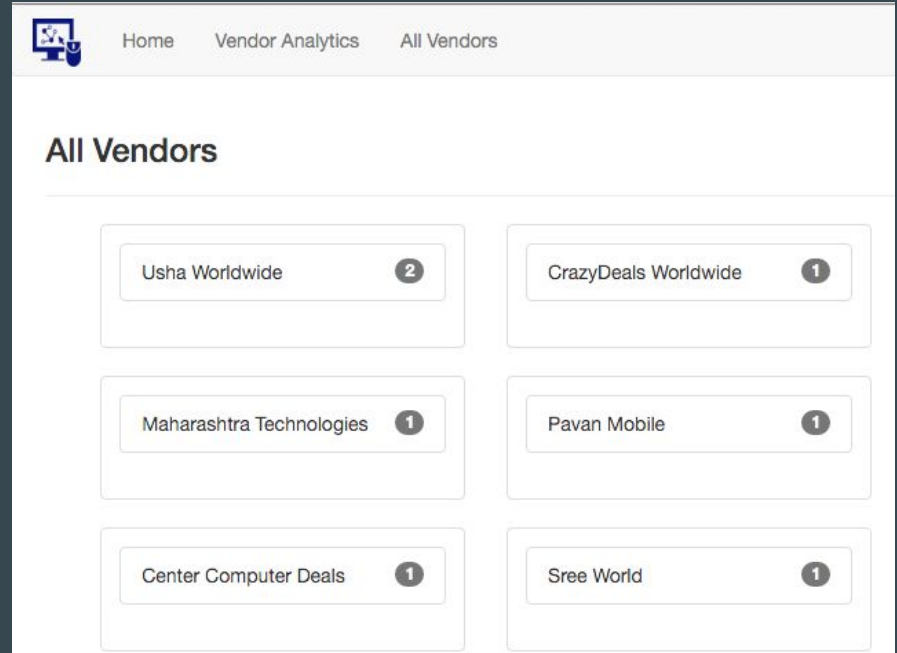
Web Application : Price stats

- Under product view we have a section for price statistics to show the variations in price for that particular product
- As we can see in the image various price variations are synced and displayed in a bar chart



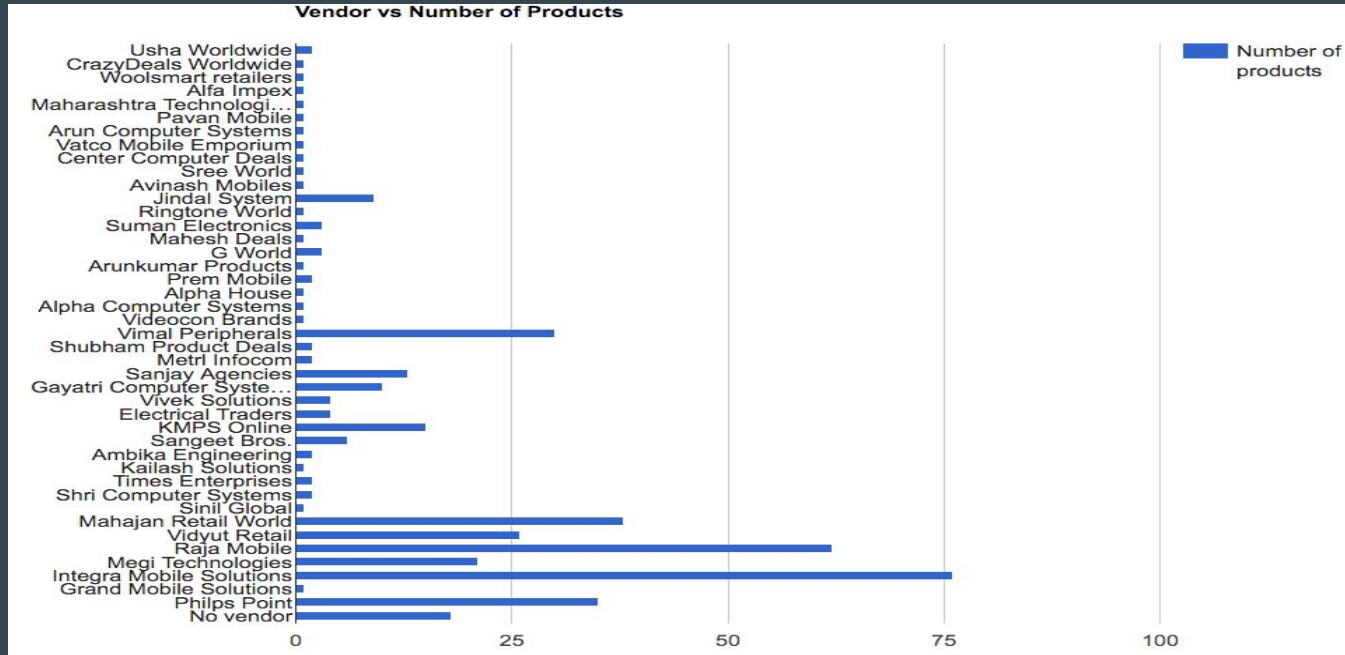
Web Application : All Vendors

- All vendors page, lists the total number of vendors that were scrapped from the website
- Also the number of products sold by each vendor is shown inside a badge
- On click of a vendor badge, the user is redirected to the list of products sold by that particular vendor



Web Application : Vendor Analytics

- Vendor Analytics page, displays a bar chart representation of vendor data i.e Vendor vs Number of products sold by that vendor



Thank you

...