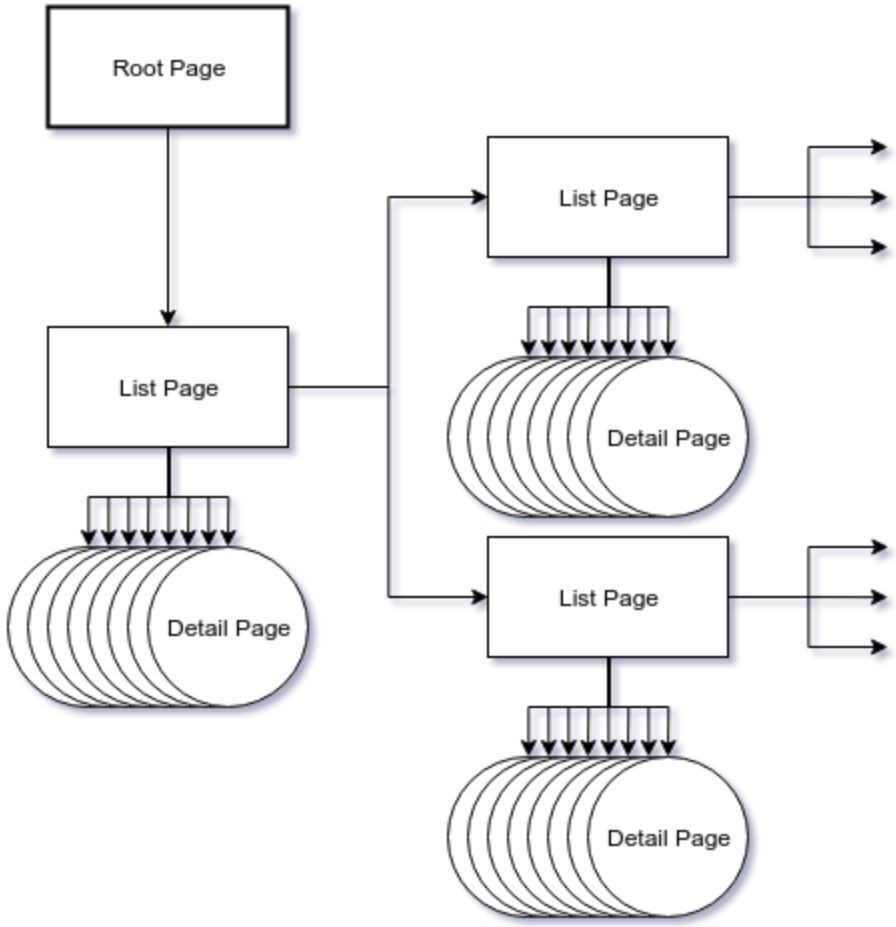
BI-VWM | VII-1 Deep Web

The objective of the thesis

The main objective of this thesis is to develop proof of concept implementation of web scraper and an API through which the scraped data can be accessed. The focus of the scraping are online bazaars and/or auctions and the ultimate goal is to build a technology which allows end user to search for a stolen item that is now being sold somewhere online. In order to achieve this goal the scraper needs to be (1) ready to scale and (2) programmed so that it can be easily extended to scrape through multiple different websites (Resources).

Basic terms

To make it easy to add new Resources let's first look into what all online bazaars or auction houses have in common, based on this a general IResource interface can then be put together. Nearly all of these websites structure their content into categories and subcategories based on a category the user will receive a set of Items. This set can then sometimes be filtered and narrowed down further. Either way at the end the website presents a first page of the result set - let's call this page the **Root Page**. This page has a unique URL (determined by the category and/or the filters) and contains (1) a list of the Items that match the query (each links to it's **Detail Page**), (2) links to one or more pages that display other items that also match the query. Let's call each page that lists multiple items (including the Root Page) a **List Page**.



Basic Terms

From the diagram above we can see that each Root Page defines a set of Items. This set will be a subset of all the items available on that Resource. Thus each Resource (e.g. sbazar.cz or cyklobazar.cz) can be divided into multiple Root Pages.

Let's store the Root Pages in a database table `root_pages`. Root Page has its `url` - that's where the scraper will start its work. Each page is also attached to a `resource` that way we'll know which pages to scrape.

Implementation - Scraper

The goal of this project is to be able to scrape many different bazaar websites. To do this efficiently let's use docker containers and have one container for each Resource. All the scraper containers will run the same image and will pick the right

implementation of the general IResource interface (adapter pattern) - this way each container will be able to scrape and parse pages of given resource.

Based on the simple analysis and description of the basic terms this is the interface that each resource needs to implement:

```
interface IResource
{

    public function getName(): string;

    public function getBaseUrl(): string;

    // Processing List Page
    public function getDetailUrls(string $listPageBody): array
;

    public function getNextListsUrls(string $listPageBody): ar
ray;

    // Processing Detail Page
    public function parseDetailPage(string $pageBody): Item;

}
```

The scraper then calls these methods in this life cycle

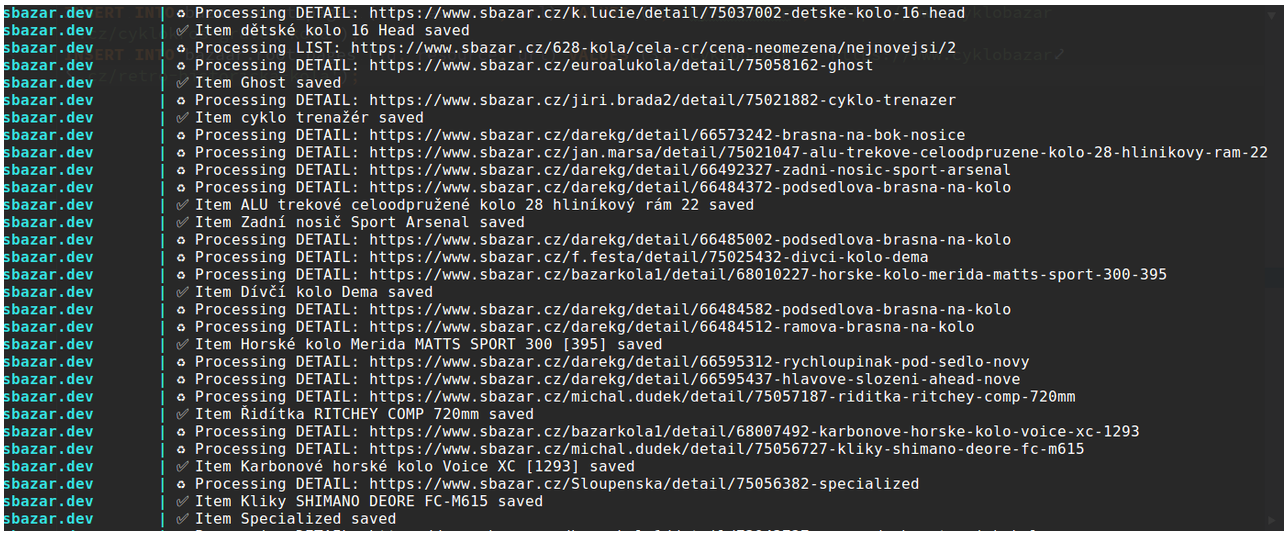
1. Fetch Root Pages from database (`root_pages`) and push them into a List Pages queue
2. Start processing a queue of requests
3. When a List Page response is received use the adapter to parse it
   a. `getDetailUrls` → push them into a Detail Pages queue
   b. `getNextListsUrls` → push them to List Pages queue

4. When a Detail Page response is received use the adapter to parse it
   a. `parseDetailPage` → Item → use storage to persist the item
5. End when there are no remaining requests in the queues

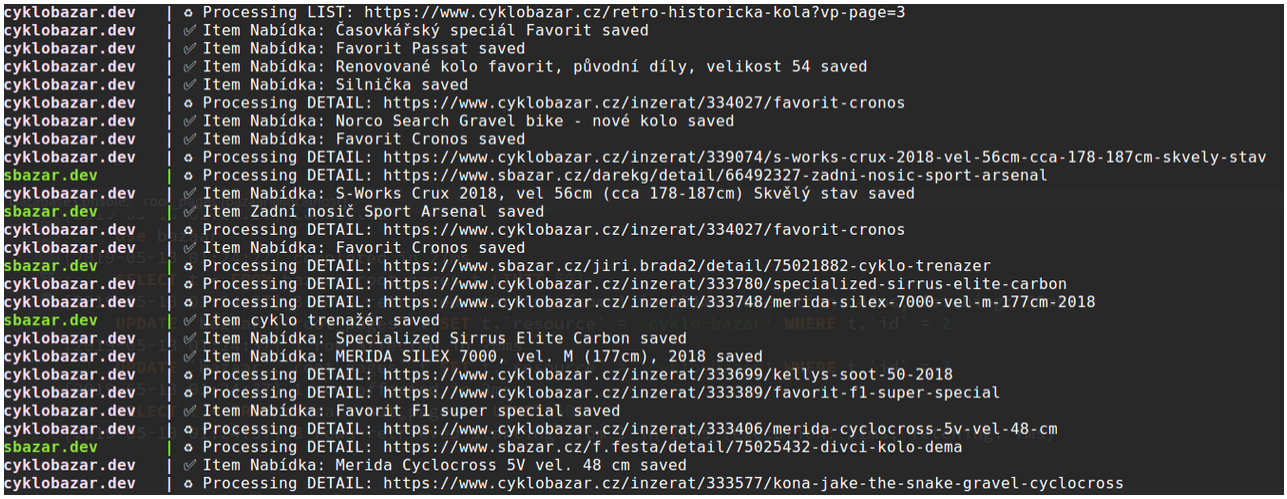
The scraper runs in PHP* and is built on top of the ReactPHP framework. Used libraries include

- symfony/Console - to crate a cli interface for the scraper (`scrape`)
- nette/DI - dependency injection container
- clue/reactphp-buzz - “simple, async PSR-7 HTTP client for concurrently processing any number of HTTP requests”
- symfony/dom-crawler
- And few others - see composer.json for more details.

In summary, there is now one image (`scraper`) which can run in multiple containers (each container can run any of the implemented Resources). Next to these containers we have a MySQL database container. To persist the data even when the whole stack gets shut down there will also be one named volume which will be used by the database.



Running one scraper container (sbazar.dev). Scraping and saving bike items from sbazar.cz.



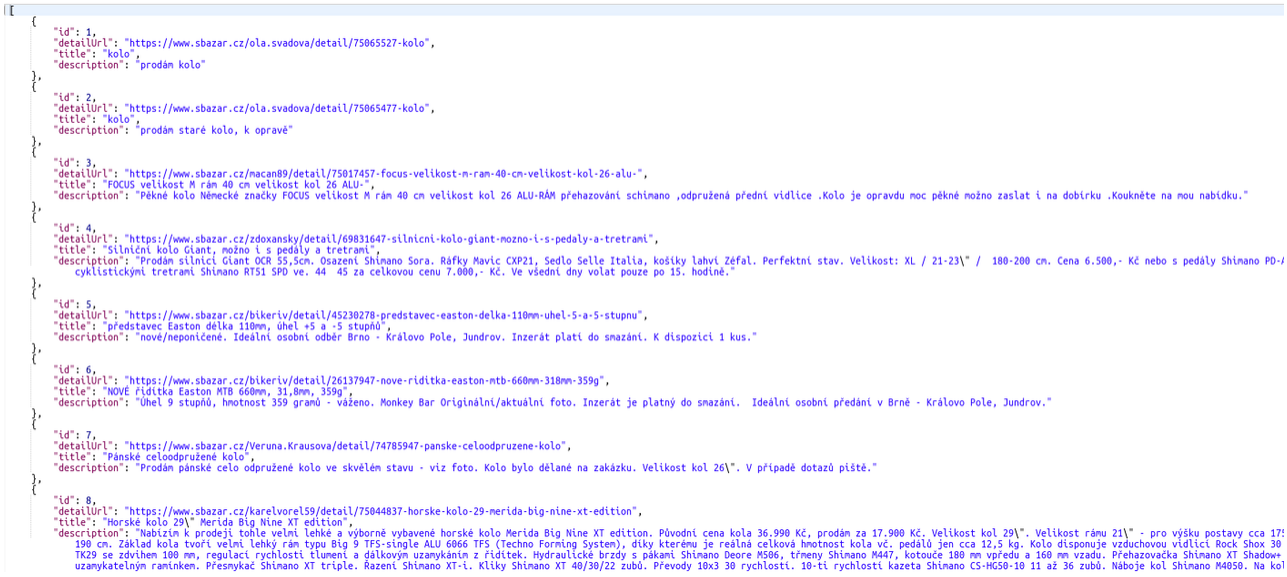
Running two scraper containers. Scraping bikes from sbazar.cz and cyklobazar.cz.

* Yes, PHP supports concurrency trying this out was one of the motivations driving this project.

Implementation - API

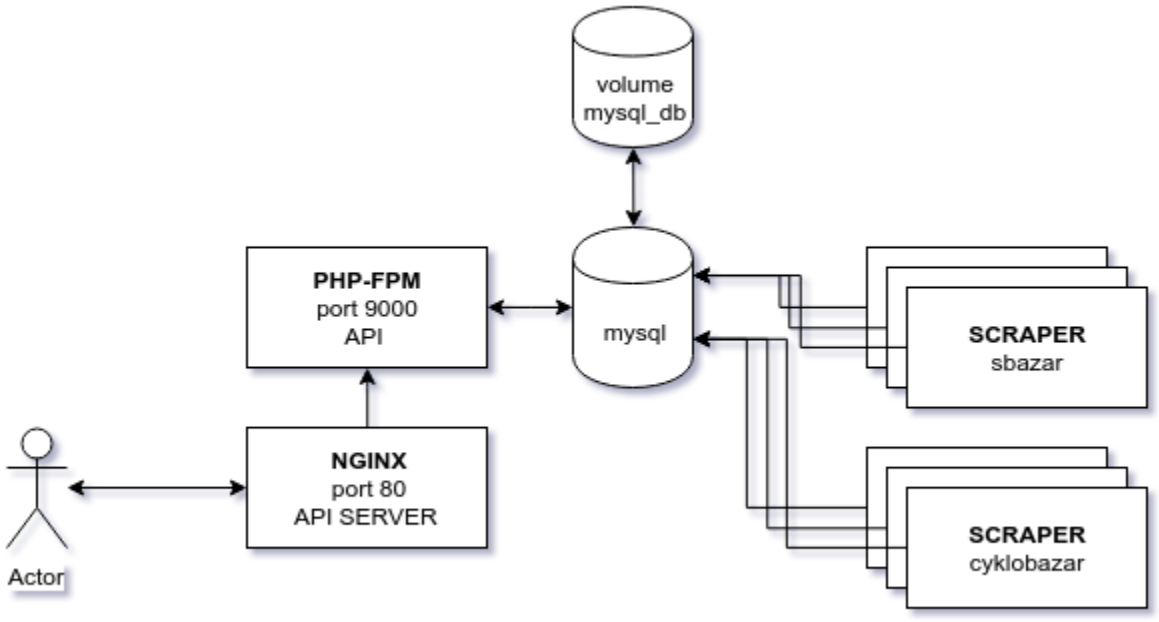
To build an API there are two more services running in this stack. PHP-FPM and NGINX web server are used for this purpose. There is be a simple Nette application which uses NextrasORM and apitte to map the data and expose a REST API.

As part of this project there only one endpoint (items) was implemented, however adding more endpoints in the future will be really simple.



Example of the api/v1/items response

Summary of the stack



Bazaar Scraper stack

Performance and throttling

The REST API is not interesting from this perspective, let's focus on the Scraping. ReactPHP abstracts away the exact mechanisms of how concurrency is achieved using it's EventLoop\Factory by default the StreamSelectLoop which makes use of the `stream_select()` method is used. This works out of the box since PHP 5.3. Quoting from the documentation of ReactPHP:

> *Under the hood, it does a simple `select` system call. This system call is limited to the maximum file descriptor number of `FD_SETSIZE` (platform dependent, commonly 1024) and scales with `O(m)` (`m` being the maximum file descriptor number passed). This means that you may run into issues when handling thousands of streams concurrently and you may want to look into using one of the alternative event loop implementations listed below in this case.*

https://reactphp.org/event-loop/#streamselectloop

EventLoop: EventLoop Component - ReactPHP •

This is the implementation used in this project and other alternatives have not been tested. However since the stack is built using docker it would be fairly simple to add the extra php extensions needed.

The main challenge of this implementation is that even though PHP supports concurrency it still runs in one thread and thus - deep down - everything is blocking. Docker containers to the rescue! As described in the previous section we can run multiple containers with the same Resource adapter. Each of the containers is independent and all of these containers can run on one host. This way asynchronicity can be achieved.

Next possible limiting factor could be the database if we're running many containers we would be keeping a lot of open connections to the database and we'd be querying the data wery often the performance could decrees. However MySQL database is very well optimised to handle this.

Other factors that limit the performance of the scraper include: the internet connection of the server running the scraper, the response time of the requested website, the speed at which the page can be parsed.

Moving onto the side of the website that is being scraped. Some websites as part of their DDoS protections etc will block IP that send too many requests (also let's be the good guys and don't abuse the target websites - there are developers on the other side). To address this issue the scraper implementation uses (in memory) queue which limits the number of concurrent requests can be sent in the same time. This can be changed in the configuration (`config.neon`) of the scraper.

Where do we go from here?

In this section let's discuss the shortcomings of this implementation and how it can be improved so that it is not just a proof of concept but a technology that can be used in the real world.

Multiple scraping modes

As described in the first section of this document, the scraper now goes into the detail page of every item. This was the most information can be retrieved, however most of the time the List Pages already contain a lot of information about each item (title, short description, price, small photo). For some use cases this could be good enough. It would probably be useful to have two different scraping modes:

1. Brief - only collect the information from the List Pages, don't query the Detail Pages at all. This will be significantly faster.
2. Thorough - query each detail page and parse the complete information from there (currently implemented)

Extend the data collected

As a proof of concept the current implementation is collecting only few parameters for each item, it would be fairly easy to extend this. The Item entity could also potentially be extended by some JSON field where all other information could then be stored. All this could then be pushed into something like elastic search for better performance.

Download images and use them for searching

This could be the killer feature of this technology. Collect the URLs of the images, have another service that will be responsible only for downloading and processing of these images. The end user would then be able to search for items based on an image. When applied to the use case of searching for stolen items online this could be really beneficial in reducing the number of items that the end user needs to go through.

In conclusion

The goal of build a proof of concept technology has been achieved. There is more work to be done to actually turn this into something that help people find their stolen bikes, but this is just the first step.

Resources

- https://sergeyzhuk.me/2018/02/12/fast-webscraping-with-reactphp/
- https://reactphp.org/event-loop

- https://github.com/apitte/core/tree/master/.docs

Antonín Vlček
vlcekan1@fit.cvut.cz
@tonyvlcek