jamk.fi

Enhancing a product's development and debugging with supporting product development

Joonas Varis

Bachelor's thesis
May 2020
Technology, communication and transport
Degree Programme in Information Technology

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Varis, Joonas	Bachelor's thesis	May 2020
		Language of publication: English
	Number of pages: 33	Permission for web publication: x

Title of publication

Enhancing a product's development and debugging with supporting product development

Degree programme

Degree programme in information technology

Supervisor(s)

Salmikangas, Esa; Huotari, Jouni

Assigned by

Valu Digital Oy

Abstract

The aim of the project was twofold. First, to develop a product which makes developing and maintaining a site search product more efficient. Second, to estimate the value of the novel supporting product development in the process of developing a site search product. The project was assigned by Valu Digital Oy, a Finnish company offering web services. The site search product the novel supporting product development aims to assist is published by Valu Digital Oy in software as a service model.

Problems the supporting product development aimed to solve were scattered debugging tools, debugging tied to personnel resource, insufficient data indexed and tools inefficient to use. The new supporting product was developed in test-driven development. The supporting product gathered product development tools to a single point of origin, enriched the log data, enabled new ways of debugging and untied personnel resources by making debugging more approachable for new developers.

The value of the supporting product development was estimated with use-cases based on problems encountered before. Each use-case execution time was timed with and without the new supporting product. Results were gathered and summarized and the return of timely investment was estimated to be around 4000 use-cases. The supporting product also offered non-quantifiable benefits, such as debugging not being tied to a personnel resource and the enablement of different development.

The project succeeded in creating graphical user interface to an abstract system. This offered new possibilities for future product development and enabled new sectors of developers to get involved in the debugging and development.

Keywords/tags

product development, test-driven development, Elasticsearch

Miscellaneous

Contents

Figures

Tables

1 Context

This chapter provides contextual information for the better understanding of the company Valu Digital Oy, its site search product, which this Bachelor's thesis aims to support, and the problematic areas of the debugging and development processes of the site search product.

1.1 Valu Digital Oy

Valu Digital Oy is a web service providing company from Jyväskylä. Founded in 1997 by Kari Turunen, who made the company's mission "auttaa asiakkaitamme toimimaan verkossa tehokkaasti ja tuottavasti" [to assist our clients in acting efficiently and productively online] (Yritys 2020).

In 2019, Valu Digital Oy's revenue was two point five million euros (Finder 2020). The company employs 30 people and it is among the leading WordPress experts in Finland. Valu Digital Oy's main source of income is customized WordPress theme implementations, hosting and further development (Virenius 2020). In addition to that, Valu Digital Oy does its own product development, such as the upcoming Headup, headless WordPress integration framework and its own site search product (Virenius 2020). The latter is described in detail in the following chapter. The name of the product is left undisclosed as per Valu Digital Oy's privacy requests, hence it is refrerred to as Site Search Product.

1.2 Site Search Product

One of the products Valu Digital Oy offers is a cloud-based site search engine, which is offered in a Software as a Service (SAAS) distributing model (Virenius 2020). Pages crawled from sites are identified with tags. Each page can have an unlimited amount of tags assigned to it that are used for result grouping.

Figure 1 presents an overview of the software stack. The site search product's software stack can be divided into three categories, i.e. frontend, backend and configuration manager, each of which are described in the subchapters below. Current development, debugging and instalment practices are also described in their respective subchapters (see 1.2.1-1.2.6). The site search product is developed and maintained by the product development team introduced in subchapter 3.1.

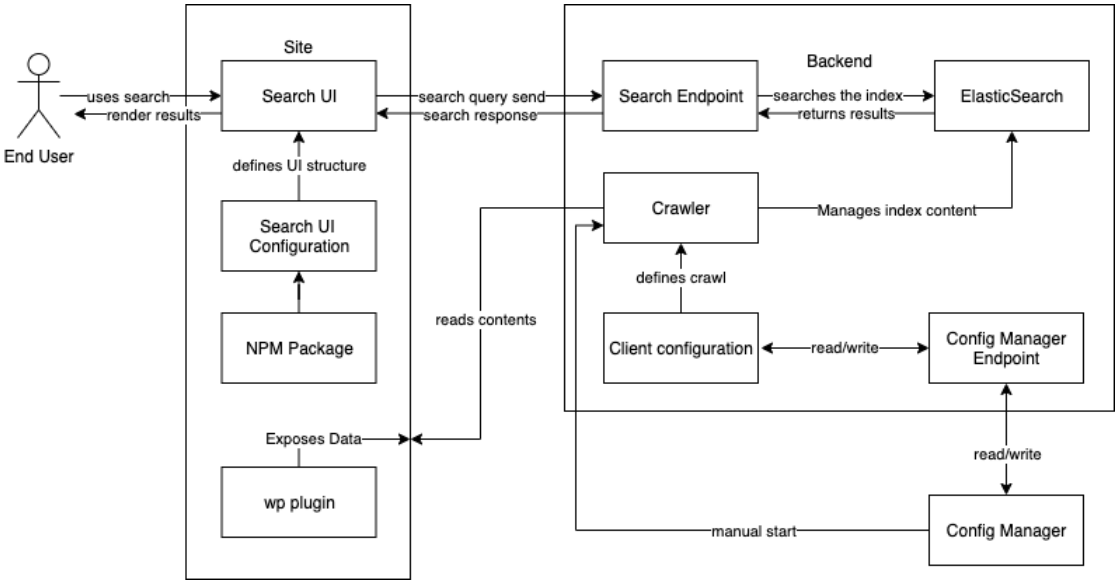

Figure 1. Site Search Product software stack

1.2.1 Frontend

Site search product's user interface is made with a npm package developed and maintained by Valu Digital Oy. The package is built with React TypeScript and offers customizable react components that can be used to build different user interfaces to match the clients' needs. Furthermore, the npm package handles sending requests to the backend and receiving responses, and rendering the data in a user-friendly way. Based on the configuration specified, the npm package sends tag queries with the search term to the backend.

1.2.2 Backend

The backend of the Site Search Product is built on a cloud platform in which its different components are run.

In the core of the Site Search Product, the data is stored in Elasticsearch database. The benefits and crude working of Elasticsearch are described in subchapter 2.7. A custom site crawler walks through the websites pages and indexes their contents, which are stored in the Elasticsearch. Crawlers can be triggered via a schedule or an endpoint. Endpoints are also used, among others, for searches and configuration saving.

1.2.3 Configuration Manager

The Site Search Product's configuration manager is a website built for managing client configurations. It can also be used for triggering manual crawls. The product development reported in this Bachelor's thesis is done as an additional feature to the Manager website.

1.2.4 Development

Each of the areas of Site Search Product is under continuous development. Because of the long planned lifespan of the product, the development is done in process akin to test driven development (TDD). TDD is described further in subchapter 2.6.

1.2.5 Debugging

Debugging the Site Search Product can be divided into two categories, i.e. customer support tickets and debugging development issues.

Customer support tickets

While the Site Search Product is in the production phase, clients who have it installed send Valu Digital Oy support tickets about unexpected product behaviour. Typical problems include:

A) Information is not found as expected

B) Wrong information is shown

Both of these can be a result from:

A) a human error during product instalment

B) a human error by client

C) a bug in the program

Debugging development issues

New features are done in TDD. Bugs are caught by tests. The cause of the bugs needs to be tracked and solved. This is complicated by the interfaces between different parts of the Site Search Product. A command line interface (CLI) is built with a series of debugging tools, but this requires a local development environment.

1.2.6 Installation

The Site Search Product is installed on a website in the following steps.

1. Creating a search configuration. Search configuration is done with configuration manager and includes information about the sites to be crawled and how the crawls are executed.

2. Crawling the sites. Sites are crawled and indexed to Elasticsearch.

3. Creating a search user interface. User Interface is created as described in the sub-chapter 1.2.1.

4. Loading the script on site. The script can be loaded from a content delivery network (CDN) with a script tag or the npm package can be installed locally.

1.3 Areas of improvement

Valu Digital Oy's product development team has identified the following five areas as problematic.

1. Inspecting indexed page is inefficient

There are tools, such as the Dejavu (Dejavu 2020), which can be used for the visual inspection of data in the Elasticsearch index. Dejavu is currently used with local Elasticsearch indexes, but using it adds steps to the debugging process. Another way to inspect indexed contents of a page is to use an application, such as Postman (Postman 2020), to send http-requests directly to the backend. Using this approach requires manual copy-pasting of customer Application Programming Interface (API) keys to request headers and manually forming search queries.

2. Debugging tools require local development environment

The site search product has helper functions for local debugging in command-line interface (CLI). Customer installation can be performed by any of the Valu Digital Oy's developers. Those outside of the product development team, do not necessarily have the site search product development environment installed. Installation and debugging should be made easy to approach.

3. Debugging is tied to personnel resource

The debugging knowhow is tied to the product development team. Although debugging is documented, it is faster to direct questions and problems to the product development team.

4. Data indexed is insufficient for debugging

Currently, the Site Search Product does not index enough metadata for debugging purposes. Data should be automatically generated from crawls and individual url failures.

5. Debugging tools are scattered

The Site Search Product's software stack is fairly complex and its different areas have their own tools for debugging. This makes installing and debugging the Site Search Product more difficult than it needs to be. All debugging tools used should be available from one point of origin.

2 Key Technologies Used

This chapter describes the seven key technologies used throughout the project.

2.1 GraphQL

GraphQL is defined as "a query language for APIs and a runtime for fulfilling those queries with your existing data" (GraphQL 2020). In GraphQL you can define a subset of data to be fetched from the data set. Queries can fetch many resources with a single request, unlike typical REST APIs (GraphQL 2020). GraphQL queries do not crash on missing values, as all the fields are nullable by default. GraphQL also offers

great developer tools and full typescript support (GraphQL 2020). An example of the GraphQL data type, query and response is depicted in Figure 2.

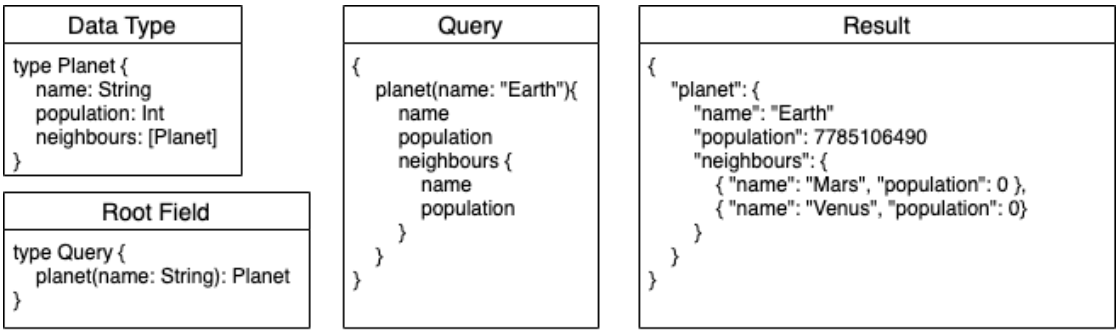

Figure 2. GraphQL data to query to results example

2.2 Cloud server services

Cloud server services is a product platform where a provider offers different services. Possible services include different virtual servers, virtual machines and more. There are currently many providers who offer these types of services, such as Amazon, Microsoft and Google (Dignan 2020).

2.3 WordPress

WordPress is a content management system (CMS) written in PHP. According to a W3Techs report on content management systems, WordPress is the leading CMS with a market share of 63.4% (CMS Report 2020).

2.4 React

React is a component based JavaScript library for UI building (ReactJS 2020), developed and maintained by Facebook (Dawson 2014).

2.5 TypeScript

TypeScript extends JavaScript by adding static type definitions, which allows using TypeScript validators while programming (TypeScript 2020).

2.6 Test-Driven Development

Test-driven development (TDD) is a type of software development process where the mindset is “test first”. In TDD, first a failing test is written, then a feature is implemented, and finally, the test is checked to pass as planned. TDD process forces the developer to divide features into smaller fragments that can be unit tested. TDD development cycle is described in Figure 3.

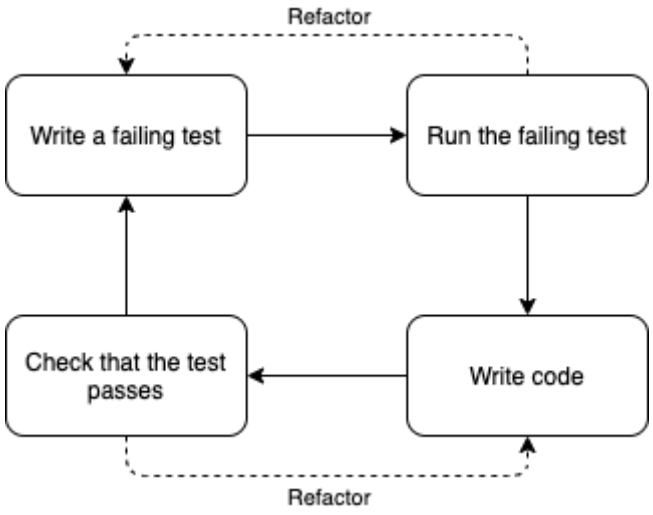

Figure 3. The development cycle of test-driven development

2.7 Elasticsearch

Elasticsearch is a document store built on Apache Lucene search engine library (Elasticsearch documentation 2020). According to DB-Engines (2020), it is the most popular database model for search engines. Elastisticsearch has many features that

specifically benefit text based search systems, like the ability to analyze the text while indexing or searching (Elasticsearch documentation 2020).

Analyzers, like snowball stemmers, can be used for making text indexes more searchable. With stemming algorithms, the system takes a word and strips it of language-specific case endings, meaning different cases of the words are searchable with other cases. For example, the English word “exciting” would be stemmed to “excit” (Snowballstem 2020).

3 Inspector

The problem areas were identified in subchapter 1.2. A supporting product to solve these problems was proposed, i.e. Inspector, which would add an interface to existing products to ease the development and debugging. Four features were identified for the minimum viable product (MVP).

1. Inspect an URL for indexed content
2. Test scrape an URL for determining how page would be indexed
3. List metadata about the individual crawls
4. List general data about the search index

The development process, implementation of each of the features and the MVP product are presented below.

3.1 Definition and development practices

The project team consisted of three members, all of whom contributed to the definition phase of the project. The team members are:

1. M. Virenius, CTO, Supervisory role in the project
2. E. Suuronen, Lead developer, Project management, definition and development

3. J. Varis, Developer, Definition, development and reporting

The Inspector was defined to be for internal use only with the possibility of exposing some of its features to the clients in the future. The technology choices were largely influenced by the technologies already used in the project. Databases were set up with Elasticsearch. Backend was written with Node.js. Frontend was written with Typescript React. Data queries were made with GraphQL. Github was used for version control.

Development was done in TDD. The alpha version of the Inspector was developed in a separate branch to a working state and merged to the master branch after code review. After that, each feature and fix was developed in a separate branch and required a code review before being merged to the master branch. Pull Requests (PRs) opened by Varis were reviewed by Suuronen and PRs opened by Suuronen were reviewed by Varis.

After identifying the four features to be included in the MVP, a wireframe model of the UI was created. The wireframe model is displayed in Figures 4 and 5.

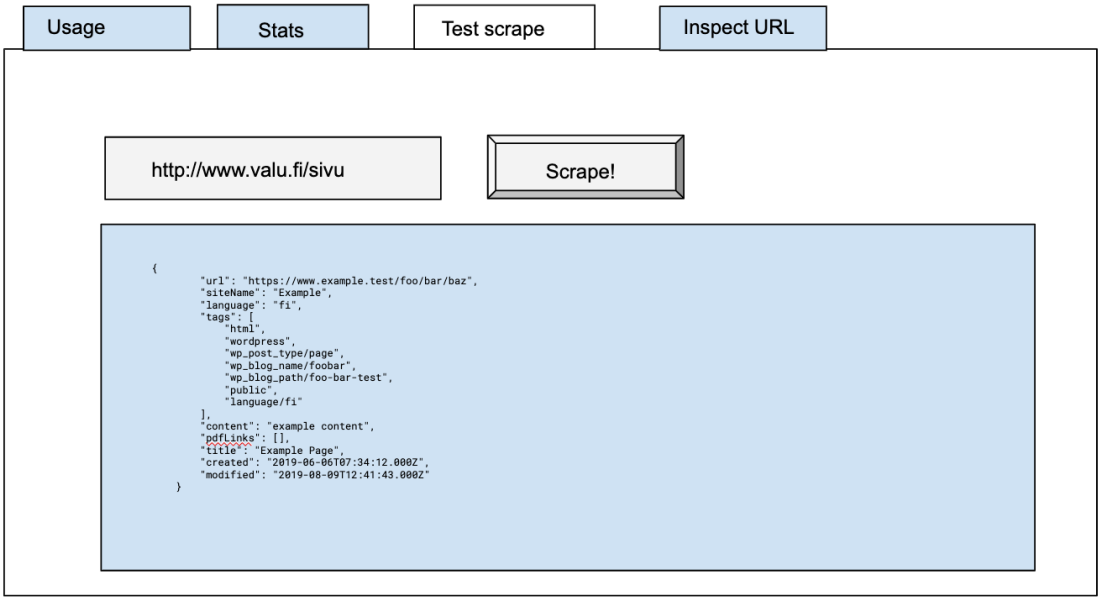

Figure 4. First wireframe UI of the Inspector, Test Scrape tab

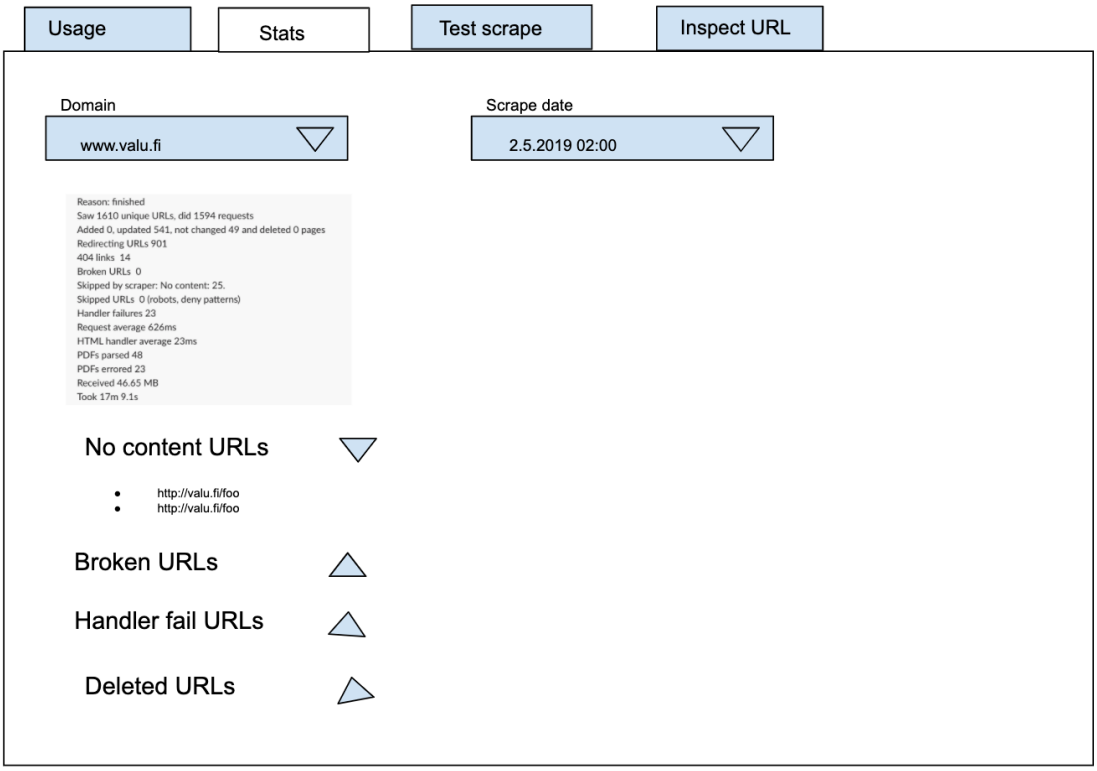

Figure 5. First wireframe UI of the Inspector, Stats tab

3.2 Requirements for Site Search Product from the Inspector definition

The features defined in the beginning of this chapter set new requirements for the Site Search Product. Each customer's search index is an individual Elasticsearch index. The need for supporting indexes was identified to fulfil the new requirements. An index where data about individual crawls were stored was added alongside the existing page index. At the same time, a page meta index was added alongside the existing page index, where information about individual page indexing was to be saved in the events when the page was not indexed in the page index.

The crawl metadata index was named “stats index”. Stats index holds the metadata of each individual crawl instance. Metadata about the site crawl offer insight into the history of the search index. The following information of the crawl is indexed: the crawl ID crawl, starting timestamp, crawl duration, the universally unique identifier (UUID) of the crawl, different domains in the index, deleted domains during the crawl, page count from sitemap(s), received bytes and an individual numeric value

for pages and pdfs (seen, added, deleted, updated, no operation and failed counts), and, lastly, a data dump field where any additional data is stored.

The page meta index was named “url meta index”. Url meta index holds information about encountered pages that were not indexed during the crawl. This data can be used to debug why a specific page was not indexed. The following information is stored for each such page: the page ID, url, domain, reason why it is not indexed, optional message, url parent links, the scrape date, UUID, used CSS-selector, used cleanup CSS-selector and a data dump field where any additional data about the rejection event is recorded.

New GraphQL resolver functions were defined to support Inspector. The different resolvers were grouped by the index they fetch data from. Methods were created for resolvers as needed. The following functions were programmed:

1. Page Index

- Fetching pages from page index
- Fetching current lock status
- Fetching current index size
- Fetching unique tags in page index
- Fetching different domains and their page counts

2. Stats Index

- Fetching an individual stat from stat index
- Fetching all the stat UUIDs
- Fetching the stat count from stat index
- Fetching the last crawl date

3. Url Meta Index

- Fetching results from url meta index

4. Other

- Performing a test scrape and returning the result without changes to the index

3.3 Final Minimum Viable Product

The product site is accessed via Configuration Manager. Each search configuration automatically gains an inspector usage page from WordPress via template hierarchy (WordPress template hierarchy).

The UI built with React TypeScript. Components are done mainly with Ant Design (2020), a React UI library.

The state of the application is stored in the URL with the history API using a react router npm package, “react-router-dom”. With this, it is possible to share the state of the React application via link, which enables efficient communication between developers working on different machines (React Router Dom, N.d.).

The new architecture with added methods is depicted in Figure 6, where the new features are colored green.

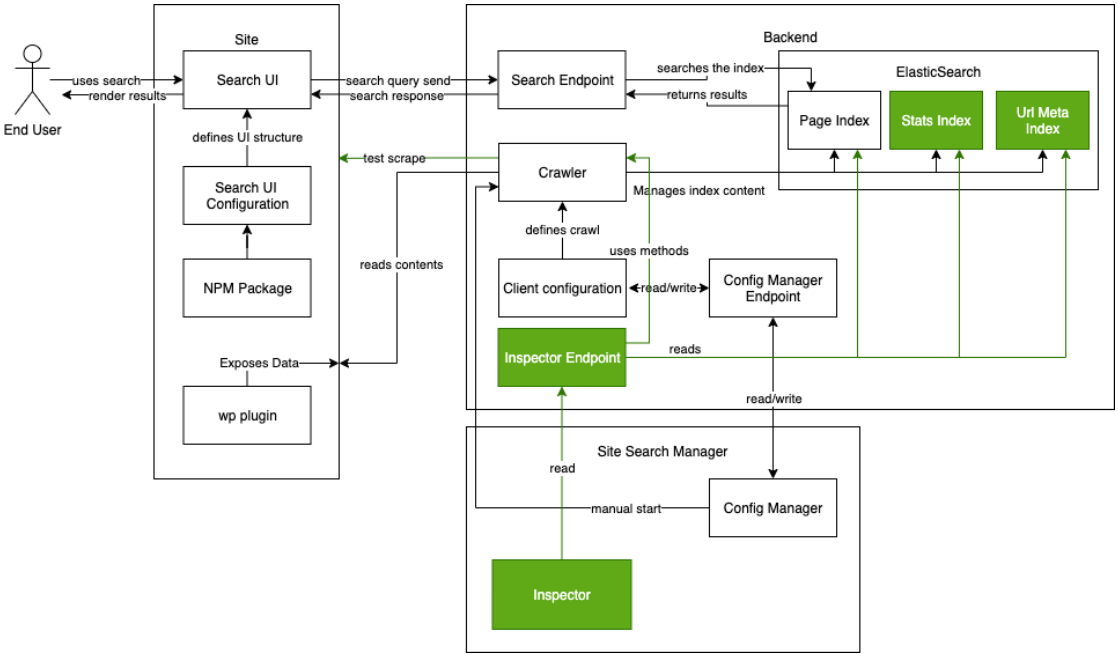

Figure 6. Site Search Product architecture figure with Inspector

When navigated through the Configuration Manager permalink in WordPress admin, the Inspector opens in the “General information” tab (see Figure 7). GraphQL queries are made to retrieve the data shown in the general tab. Detailed GraphQL queries made for each of the Inspector tabs are outlined at the end of subchapter 3.2.

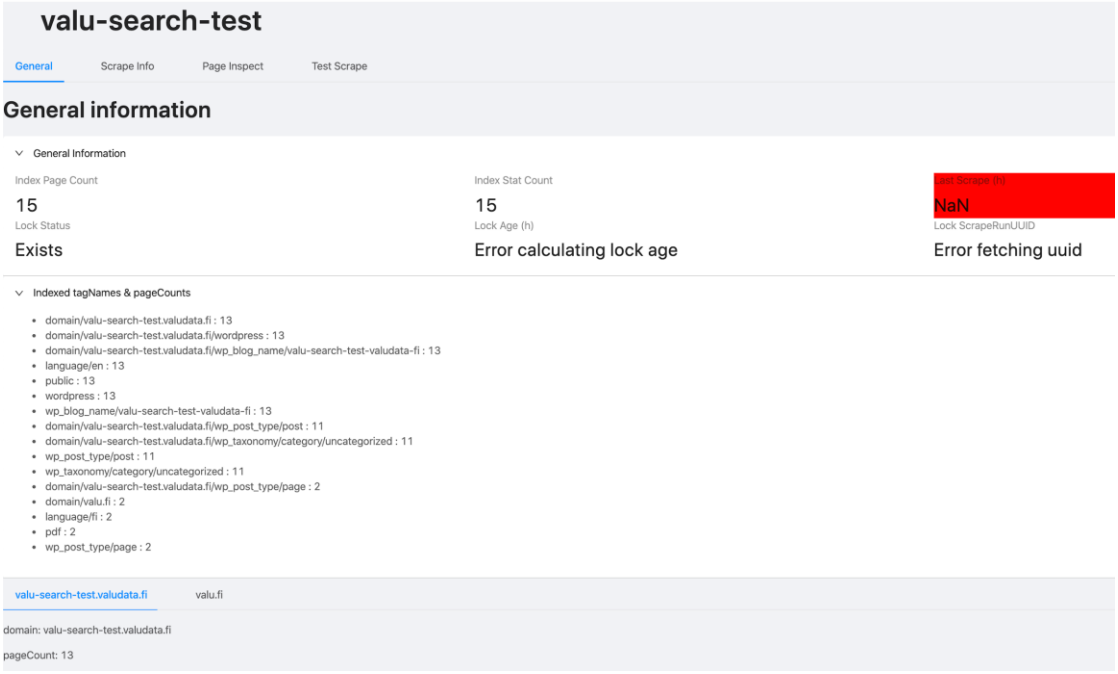

Figure 7. Inspector General information

The second tab, “Scrape Info”, displays metadata about the specific crawl instance and page meta associated with said crawl, shown in Figures 8, 9 and 10. Crawl instances can be switched with the dropdown menu at the top part of the page, as demonstrated in Figure 9. Page metadata for individual targets can be browsed at the bottom of the page. After choosing the target, the user can browse through the grouped pages. An example of a skipped link entry can be seen in Figure 10.

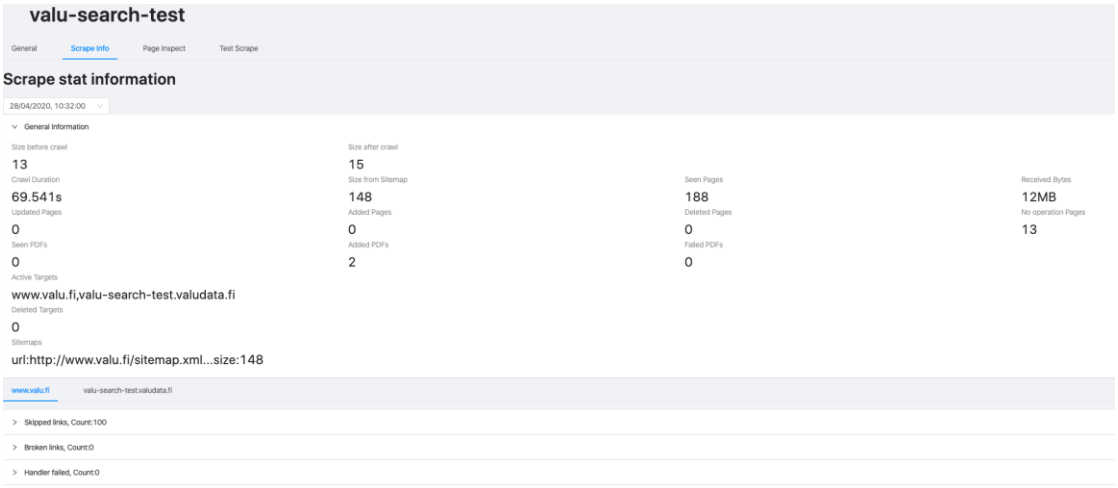

Figure 8. Scrape Info main view

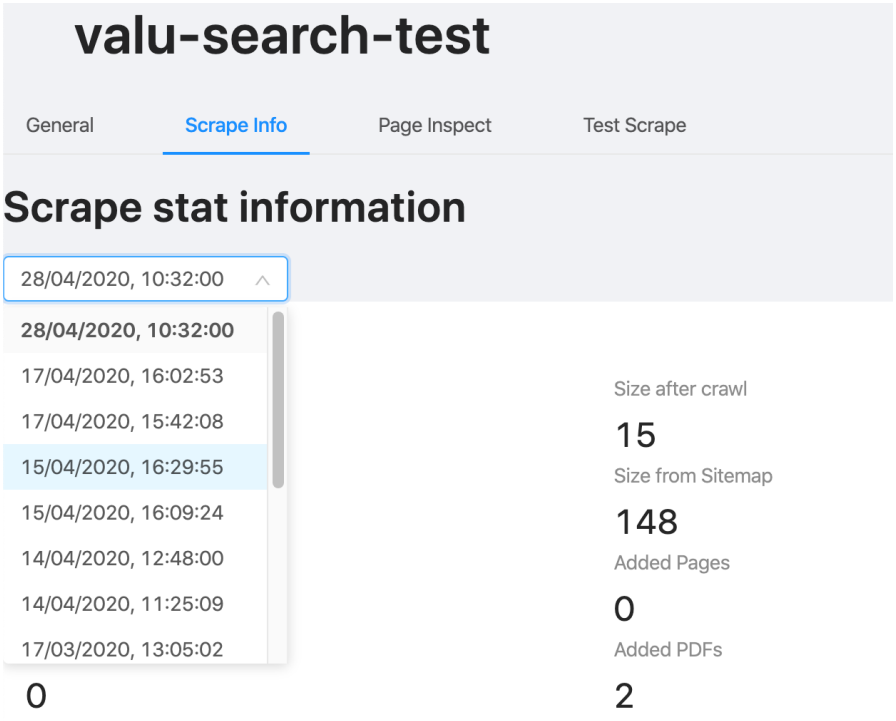

Figure 9. Scrape Info dropdown menu

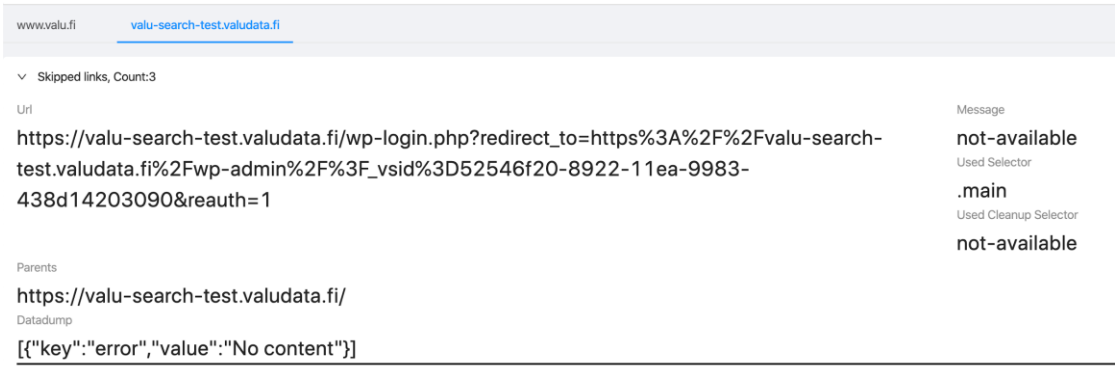

Figure 10. Example of a skipped link

On the third tab, “Page Inspect”, the user can inspect indexed data about any given url. If the page was not indexed for some reason, the GraphQL query fallbacks to fetching from url meta index. GraphQL enables resolving different return types based on the return type. When searching for a page, the Inspector first tries to search for page in page index. If no result is found, the last crawl’s UUID is used to match to results from url meta index. The returned data are provided in JSON format that exposes the page index mapping and is redacted (see Figure 11).

Figure 11. Page inspect, url meta index

On the fourth tab of the Inspector, “Test Scrape”, the user can perform test scrapes on any given url without making changes to the indexes. The test scrape feature scrapes the page and returns the same data the same way as the actual crawl process. This is achieved by using the same functions and methods in the backend. The Test scrape handles both pages that would be and would not be indexed and provides information on both. If the page were to be indexed, the method would return all the data that would be indexed in the same format that the page index uses. If the page were to be indexed in url meta index, the method would return the data in same format the url meta index uses. Data models were deemed sensitive and were redacted from figure 12.

Figure 12. Test Scrape

4 Inspector Applications in Product Development

The four features chosen for the MVP solution each address a problematic area identified in subchapter 1.2. These features introduce new ways for debugging, while they also improve and collect existing ones under the same point of origin.

4.1 Estimating Inspector value

Currently, there is no data set large enough to perform scientific analysis on before or after the implementation of Inspector. However, the measured benefit of the Inspector can be estimated on a use-case basis by listing steps and the time it takes to perform each use-case with and without the Inspector. Use-cases selected for analysis are chosen optimistically, i.e. they directly benefit from a feature implemented in the Inspector. The Site Search Productcan be installed on sites other than WordPress, but only WordPress sites hosted by Valu Digital Oy with a wp-valu-search plugin installed are used in demo scenarios. The use-cases were tested in a manner that the user knows what to do next and how to perform each of the steps. Each of the steps involved in the use cases were performed and timed by Varis in precision of five seconds.

Some of the problems identified in subchapter 1.3 are not easily quantifiable, such as “Debugging is tied to personnel resource”. An estimation of the value of improvements to the identified problems was beyond the scope of this thesis.

4.2 Use-cases

Use-cases are described with how a client or developer would experience product behaviour. Then, the underlying causing effect is listed and an explanation why a particular behaviour occurs is provided. Steps to identify the underlying cause and the time to execute those steps are listed with and without the use of Inspector.

4.2.1 Case 1. Page is not found with exact title

Cause: Faulty content selector.

Explanation: Developers and clients can define page content CSS-selectors and cleanup CSS-selectors. It is possible to make human error with overlapping or faulty selectors.

Steps without Inspector:

1. Navigating to the clients page (5s)
2. Use the client's search to rule out user error (10s)
3. Logging in to the clients WordPress admin panel (15s)
4. Checking the page has not been published since the last crawl (15s)
5. Navigating to the page in question (5s)
6. Checking the page meta tag wp-valu-search exposes (20s)
7. Combining content selectors and cleanup selectors manually from Configuration Manager and page meta tag (60s)
8. Manually checking the page's DOM structure (120s)

total: 250s

Steps with Inspector:

1. Open Configuration Manager (15s)
2. Open clients inspector (5s)
3. Navigate to page inspect (5s)
4. Input page's url in question → error: no content is shown + used content selector and cleanup selector (5s)
5. Manually checking the page's DOM structure (120s)

total: 150s

4.2.2 Case 2. Page is not found with exact title

Cause: Require meta tag option is set to true, and wp-valu-search is disabled.

Explanation: By default, the site search product only indexes pages that exposes a meta tag, which provides additional information about the page to crawl. When wp-valu-search is not installed or the plugin is disabled from wp-admin, no meta tag is shown.

Steps without Inspector:

1. Navigating to the clients page (5s)
2. Use the client's search to rule out user error (10s)
3. Logging in to the clients WordPress admin panel (15s)
4. Checking the page has not been published since the last crawl (15s)
5. Navigating to the page in question (5s)
6. Checking the page meta tag wp-valu-search exposes (5s)

total: 55s

Steps with Inspector:

1. Open Configuration Manager (15s)
2. Open clients inspector (5s)
3. Navigate to page inspect (5s)
4. Input page's url in question → error: "No metatag provided" (5s)

total: 30s

4.2.3 Case 3. Page is not found with exact title

Cause: There is no link to the page and page is missing from sitemap.

Explanation: The site search product crawls through websites by walking links and / or using the sitemap(s). If the site does not have a link leading to the page and it's not present in the sitemap the page cannot be found without defining it as a starting path.

Steps without Inspector:

1. Navigating to the clients page (5s)
2. Use the client's search to rule out user error (10s)
3. Logging in to the clients WordPress admin panel (15s)
4. Checking the page has not been published since the last crawl (15s)
5. Navigating to the page in question (5s)

6. Checking the page meta tag wp-valu-search exposes (20s)
7. Using CLI tools from local development environment for scraping the page (15s)
8. Crawling the site to local elasticsearch index (120s)
9. Checking the local Elasticsearch index for page in question (60s)
10. Checking sites sitemap manually for missing page (30s)

total: 325s

Steps with Inspector:

1. Open Configuration Manager (15s)
2. Open clients inspector (5s)
3. Navigate to page inspect (5s)
4. Input page's url in question → no page in index message (5s)
5. Navigate to test scrape (5s)
6. Input page's url in question → shows page correctly (5s)
7. Checking sites sitemap manually for missing page (30s)

total: 70s

4.2.4 Case 4. Faulty data is shown in search results

Cause: Faulty content and or cleanup CSS-selectors.

Explanation: Developers and clients can define page content CSS-selectors and cleanup CSS-selectors. It is up to the developer installing the site search product to select correct content/cleanup CSS-selectors.

Steps without Inspector:

1. Navigating to the clients site (5s)
2. Using browser developer tools correct content / cleanup CSS-selectors are determined for different page layouts. (300s)
3. Open Configuration Manager (15s)
4. Update client's Configuration (15s)
5. Crawl site with updated config (300s --- 8h)

6. Search results are checked again and CSS-selectors are updated if necessary. (120s) → This is problematic as it might take many iterations to get selectors to be perfect. Each crawl can take anywhere from 5 minutes to many hours depending on the size of the site.

total: 555s + (crawl time x N)

Steps with Inspector:

1. Navigating to the clients site (5s)
2. Using browser developer tools correct content / cleanup CSS-selectors are determined for different page layouts. (300s)
3. Open Configuration Manager (15s)
4. Update client's Configuration (15s)
5. Open clients Inspector (5s)
6. Navigate to test scrape (5s)
7. Ensure correct selectors (120s)
8. Iterate if necessary
9. Crawl site with updated config (300s --- 8h)

total: 465s + (iterations * N) + crawl time

4.2.5 Case 5. Developer selecting tags for search UI configuration

Cause: Tags are used for content grouping as described in subchapter 1.1.

Explanation: When defining the site search product's user configs developers need to form tag queries which are used for content grouping.

Steps without Inspector:

1. Navigate to CDN endpoint exposing client specific tags in JSON-format (20s)
2. Select correct tags from JSON (120s)

total: 140s

Steps with Inspector:

1. Open Configuration Manager (15s)
2. Open clients Inspector (5s)
3. Click to open indexed tags and page counts to see tags in used friendly format (5s)
4. Select tags (60s)

total: 85s

4.3 Estimating Inspector value based on use-case results

In all of the selected cases, the Inspector saved time. Results of the use-case study are shown in table 1.

Table 1. Inspector use-case summary

Inspector use-case summary				
Case	Completion time without Inspector(s)	Completion time with Inspector(s)	Time Saved(s)	% time saved / without inspector
Case 1. Page is not found with exact title	250	150	100	40.00%
Case 2. Page is not found with exact title	55	30	25	45.45%
Case 3. Page is not found with exact title	325	70	255	78.46%
Case 4. Faulty data is shown in search results*	555	465	90	16.22%
Case 5. Developer selecting tags for search UI configuration	140	85	55	39.29%

* the final time of the case largely depends on the customer crawl time

If we assume that the average use-case matches the average of cases selected, the time saved with Inspector on an average use-case equals the average time saved of use-cases (105 seconds). The Inspector took approximately 116 hours to complete. The Inspector's return of investment on use-cases alone would take 3977 cases.

Nearly 4000 cases sounds like a lot, but each individual installation instance includes many instances where the Inspector is used. The developer installing the search product uses the Inspector approximately 8 to 10 times and when the developer skips one of these steps, it tends to result in a support ticket in the future.

The benefits of the Inspector in the Site Search Product's development are more polarized as the Site Search Product's technology stack is wide and the Inspector only covers a very specific area of it. Therefore, while it might be invaluable in some of the development, it might not be used at all in others.

5 Discussion

The novel project described in this thesis supports that the product development process can be enhanced with supportive product development. The value of said product development must be evaluated on a case-by-case basis. It should be noted that the value of supporting product development should not be judged by return of investment over use-cases alone, as it can offer immaterial benefits which are hard to quantify. Regarding the Inspector project, such benefits were debugging is no longer tied to personnel resources, enrichment of data logs that offer a basis for new ways of debugging, and professional growth and learning gained during the project by the project team. The Inspector also adds value when it comes to training employees new to the Site Search Product. If the Site Seach Product stack is easier and faster to comprehend, hours or even days are cut out of the time it would take to reach the same level of competence with the system.

Currently, approximately one fourth of the developers at Valu Digital Oy have installed the Site Search Product for a client and first installations were done without the Inspector. The general consensus is that the current documentation and the developer tools developed have made installation relatively simple and easy to follow. Virenius estimates that the number of developers who have installed the Site

Search Product by the end of 2020 will rise to nearly 100% (Virenius 2020). Future use and wider user base will determine the true value of the Inspector.

During the development of Inspector, the groundwork for building a visualization into an abstract system was done. This enables personnel outside the Site Search Product development team to gain independent insights about the system. In the future, Valu Digital Oy plans to add more tools to the Inspector for its developers and plans to keep the option to expose some part of the Inspector to its clients. The development of the Inspector has opened new doors for product development and offers a solution for the problematic areas identified in the beginning of the project.

References

Ant Design. 2020. *React UI library*. Accessed 14.4.2020. https://ant.design/

Dawson, C. 2014. *JavaScript's History and How it Led To ReactJS*. Web article published in the new stack website. Accessed 14.5.2020. https://thenewstack.io/javascripts-history-and-how-it-led-to-reactjs/

DB-Engines. 2020. *DB-Engines Ranking of Search Engines*. Monthly updated web chart about dbms popularity. Accessed 14.5.2020. https://db-engines.com/en/ranking/search+engine

Dejavu. 2020. *Appbase's Open Source library; Dejavu*. Accessed 2.5.2020. https://opensource.appbase.io/dejavu/

Dignan, L. 2020. *Top cloud providers in 2020*. Web article published in zdnet, business technology news website. Accessed 22.5.2020. https://www.zdnet.com/article/the-top-cloud-providers-of-2020-aws-microsoft-azure-google-cloud-hybrid-saas/

Elasticsearch documentation. 2020. *Documentation of Elasticsearch*. Accessed 14.5.2020. https://www.elastic.co/guide/en/elasticsearch/reference/current/search-analyze.html

Finder. 2020. *Finder, Financial information*. Accessed 24.4.2020. https://www.finder.fi/Internet-palvelut/Valu+Digital+Oy/Jyv%C3%A4skyl%C3%A4/yhteystiedot/156790

GraphQL. 2020. *GraphQL technology website*. Accessed 2.5.2020. https://graphql.org/

Postman. 2020. *Postman application website*. Accessed 2.5.2020. https://www.postman.com/

ReactJS. 2020. *React technology website*. Accessed 22.5.2020. https://reactjs.org/

React Router Dom. N.d. *Npm package documentation website*. Accessed 14.4.2020. https://reacttraining.com/react-router/web/api/Hooks/usehistory

Snowballstem. 2020. *Snowball string processing language website*. Accessed 14.5.2020. https://snowballstem.org/

TypeScript 2020. *TypeScript technology website*. Accessed 14.5.2020. https://www.typescriptlang.org/

Usage statistics of content management systems. Report about CMS usage percentile. W3Techs. Updated daily. Accessed 22.5.2020. https://w3techs.com/technologies/overview/content_management

Virenius, M. 2020. CTO. Valu Digital Oy. Interview. 1.11.2020.

WordPress template hierarchy. N.d. *WordPress documentation*. Accessed 14.5.2020. https://developer.wordpress.org/themes/basics/template-hierarchy/

Yritys. 2020. Valu Digital website. Accessed 24.4.2020. https://www.valu.fi/yritys/