

INFORMATION SCIENCE

Master Thesis

Semantic Web Application for Email Receipts

Author: Fredrik Otterlei Madsen

Supervisor: Csaba Veres

1 June 2018

Abstract

This thesis is about enhancing receipts by extracting information from them and create receipts in semantic format. It involves data mining techniques for processing emails and categorisation of products. These techniques and methods were implemented and tested in a web application, which confirms that it is possible to create receipts in a semantic format. External services were used to analyse existing semantic metadata for products and perform category search. The findings showed that there is improvement potential in how receipts are modelled, especially with regard to semantic category specification of products.

Acknowledgements

I want to thank friends and fellow students for a memorable year with interesting work, discussions and fun moments. They have motivated me to complete this thesis. I would also like to thank my supervisor Csaba Veres for his guidance and help during the work on this thesis.

Contents

Abstract	I
Acknowledgements	II
1 Introduction	1
1.1 Research Questions	3
2 Theory	4
2.1 E-commerce	4
2.2 RDF and Semantic Web	5
2.3 Regular expressions	6
2.4 Schema.org and GoodRelations	8
2.5 Google Knowledge Graph	11
2.6 Data visualization	14
3 Research Method	16
3.1 Design Science Research	16
3.2 Development Methodology	18
3.3 Evaluation	19
3.3.1 Statistical Methods	19
3.3.2 Metrics	21

4	Development	22
4.1	Tools	22
4.2	Artifact	23
4.3	Planning	24
4.4	Data Mining: Parsing Emails	26
4.4.1	Regular Expressions	28
4.5	Data Modelling: Schema.org	29
4.6	Classification	34
4.7	Data Visualization: Sgvizler2	38
5	Analysis	45
5.1	Data mining and semantic modelling	45
5.2	Product classification	46
5.3	Product metadata	47
6	Discussion	50
6.1	Data extraction technique	50
6.2	Adoption of Schema.org vocabulary	52
6.3	Classification of products	55
7	Conclusion	58
7.1	Research questions	58
7.2	Future work	59
	Appendix A Subject Filter and Table Search	64
	Appendix B Regular Expression	67
	Appendix C Organizations JSON-LD	73
	Appendix D Classification Google knowledge graph	76
	Appendix E Metadata from products	78

List of Figures

3.1	Conceptual model of the iterative cycle	18
3.2	Precision and Recall equations (Wimalasuriya, 2010, p. 318) .	21
3.3	F-measure equation (Wimalasuriya, 2010, p. 318)	21
4.1	Trello Board retrieved from https://trello.com	25
4.2	JSON receipts from data mining	30
4.3	Receipt in Turtle format	31
4.4	Email identification of organizations	32
4.5	Response from Google knowledge graph search	36
4.6	Schema.org definition of type Book	37
4.7	Dashboard view from localhost:8080/dashboard	38
4.8	Receipt view from localhost:8080/dashboard/1001	39
4.9	Pie chart category types from brokers made in Sgvizler2	41
4.10	Pie chart for items made in Sgvizler2	42
4.11	Bar chart made in Sgvizler2	43
4.12	Line chart made in Sgvizler2	43
4.13	Trendline chart made in Sgvizler2	44
5.1	Metadata product results	49
6.1	GenTax example	56

List of Tables

3.1	Hevner et al. (2004)/Design Science in IS Research	17
4.1	How the parser reads a table	27
4.2	Composition of key attributes and data values	27
4.3	Regular Expressions for total price and currency	29
5.1	Resources and properties from knowledge graph	46
5.2	Types classified for items through Google knowledge graph . .	47

Acronyms

API Application Programming Interface. 30

CSS Cascading Style Sheets. 33

HTML HyperText Markup Language. 4, 12, 33, 40

JSON JavaScript Object Notation. 12

JSON-LD JavaScript Object Notation for Linked Data. 6, 30, 32, 34

OWL Web Ontology Language. 3

RDF Resource Description Framework. 3, 9, 32

RDFa Resource Description Framework in Attributes. 4, 6, 7

RDFS Resource Description Framework Schema. 3, 32

SPARQL Simple Protocol and RDF Query Language. 12, 16, 17, 33, 34, 40,
41

URI Uniform Resource Identifier. 3

XML Extensible Markup Language. 12

XSD XML Schema Definition. 41

Chapter 1

Introduction

The research in this thesis is about how to implement and use semantic vocabularies to represent digital receipts. Receipts are transitioning to electronic platforms and data formats, such as web services and smartphone apps delivering digital receipts to customers. Digital receipts highlight information about our purchases, by using semantic technologies such as vocabularies and classification searches it is possible to present and identify the things we buy. In this research project I have explored the format of digital receipts and developed a proof of concept web application that can lift data to a reasonable semantic level. This will help keep track of recent purchases and improve processing of receipts.

The artefact I planned and developed as part of my thesis is a web application. It runs in the Node.js environment and contains a client-side for the visual representation for users, and a server side for routes, functions, models, data retrieval and exchange with semantic services. The semantic layer runs in Fuseki (Apache, 2017), a semantic software service for producing and querying semantic knowledge graphs. Data is retrieved from emails in Gmail, emails are sent from various web stores and service providers. Furthermore, a function parses each email and reproduce them as semantic resources which are inserted in the Fuseki endpoint.

The main visual features of the web application is to show things a user has bought over time, details about receipts and graphs. Receipts are presented in a list on the dashboard, where each unique receipt is clickable to view more details. There is also a separate web page for graphs showing statistics from receipts.

1.1 Research Questions

The following research questions are stated:

- Q1. Is there sufficient metadata in web resources to categorise receipts?
- Q2. What sort of categories are available in web markup for products and services?
- Q3. What other methods can be used to enhance the category structure?
- Q4. Can we use the categories in aggregations and visualization?

Chapter 2

Theory

2.1 E-commerce

Electronic commerce is related to purchase and sale of goods and services on the World Wide Web. It involves several types of agents and generates a large work flow. A trade is an agreement between seller and buyer, but the trade itself involves parties that handle involved actions. These actions are for instance information flow, brokerage, money transaction, and transportation. Liu et al. (2015) highlight the importance of connecting online trading and logistic services in their research paper. They mention electronic warehouse receipts for creating and updating information before, during, and after delivery of goods. "The pattern of e-commerce trading based on electronic warehouse receipts has the characteristics of digitalization and standardization, which can help the e-commerce platform achieve seamless docking with logistics and have high technology feasibility" (Liu et al., 2015, p. 662).

The proposed system structure for e-commerce is based around the online trading platform and it can be divided into five parts which are the e-commerce platform, traders, delivery warehouse, logistics service providers and balance bank. The e-commerce platform is an online trading platform responsible for organizing and regulating transactions. It is capable of controlling and sharing

information for involved participant on the demand side. Liu et al. (2015) conclude that an e-commerce system with integrated logistics services will provide efficient transactions and guarantee the quality of trading products.

In relation to this thesis, the e-commerce system that Liu et al. (2015) propose is relevant in terms of modelling agents, products, services, sales, payment, receipts and logistics. It is a comprehensive system that covers many entities, compared to this thesis which focus on receipts. However, we find the same entities in the Schema.org vocabulary, which shows that there exist tools that would fit in an e-commerce system.

2.2 RDF and Semantic Web

Resource Description Framework (RDF), Resource Description Framework Schema (RDFS) and Web Ontology Language (OWL) are modelling languages of the Semantic Web. These languages are used for adding additional meaning to data, a process known as describing data and referred to as semantic metadata. The foundation is RDF, a standard model for distributing data on the Web according to W3C (2014). This model structure is also mentioned and specified as a triple, the fundamental data structure of RDF. A triple consist of a subject, predicate and object. It has two nodes, the subject and object linked together with the predicate (subject-predicate-object). RDF can describe all sort of things in the world, "things" are often referred to as resources. Each resource has a Uniform Resource Identifier (URI), Berners-Lee (2006) specified 4 rules that are expectations of behaviour for the Semantic Web:

1. Use URIs as names of things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).

4. Include links to other URIs so that they can discover more things.

Resource Description Framework in Attributes (RDFa) is a markup language that makes statements about resources in the form of subject-predicate-object expressions known as triples. It is implemented in XHTML as a core library and state explicit rules on how to define attributes for embedding semantic markup in host languages (W3C, 2015). An example is RDFa generated in HTML documents, which give additional meaning in tags that contain data. Sheth & Thirunarayan (2012) provide a clear definition of RDFa:

The core subset of RDFa attributes include

- *about* - a URI extracted as the subject of an RDF triple that specifies the resource the metadata is about;
- *rel* and *rev* - extracted as the object property (predicate) of an RDF triple, this URI specifies a relationship or reverse-relationship with another resource;
- *href*, *source*, and *resource* - extracted as the object of an RDF triple, this URI specifies the partner resource;
- *property* - extracted as the datatype property (predicate) of an RDF triple, this URI specifies a property for the content of an element; and
- *instanceof* - extracted as the object property 'rdf:type' coupled with and RDF triple's object, this optional attribute specifies the RDF type of the subject;

2.3 Regular expressions

A regular expression is capable of processing text by searching a pattern. It is a concept from theoretical computer science and formal language theory,

proposed by Stephen Kleen in 1956. Since then, it has been implemented in various systems and programming languages. "The basic function of regular expressions are matching, substitution and extraction" (Bakar, 2014). In terms of data mining, it is a fitting technique for extracting and processing data from text, especially since it does not require heavy processing.

JavaScript is a programming language that supports regular expressions, from the syntax it is possible to use both regular expression literals and objects. Regular expressions are composed by a large set of characters representing logical patterns. Here are some of the often used characters with explanation from gskinner (2018):

Character classes

.	any character except newline
\w \d \s	word, digit, whitespace
\W \D \S	not word, digit, whitespace
[abc]	any of a, b, or c
[^abc]	not a, b, or c
[a-g]	character between a & g

Anchors

^	start of string
\$	end of string
\b \B	word, not-word boundary

Escaped characters

\. * \\	escaped special characters
\t \n \r	tab, linefeed, carriage return
\u00A9	unicode escaped

Groups & Lookaround

(abc)	capture group
\1	backreference to group #1
(?:abc)	non-capturing group
(?=abc)	positive lookahead
(?!abc)	negative lookahead

Quantifiers & Alternation

<code>a*</code>	<code>a+</code>	<code>a?</code>	0 or more, 1 or more, 0 or 1
<code>a{5}</code>	<code>a{2,}</code>		exactly five, two or more
<code>a{1,3}</code>			between one & three
<code>a+?</code>	<code>a{2,}?</code>		match as few as possible
<code>ab cd</code>			match ab or cd

2.4 Schema.org and GoodRelations

Schema.org is a vocabulary that promote schemas for structuring data on the Internet, it support encodings like RDFa, Microdata and JSON-LD. The history between Schema.org and GoodRelations goes back to 2010 when Google adopted the GoodRelations vocabulary according to Hepp (2015, p. 726). The Schema.org project is a collaborative community founded by Google, Yahoo, Microsoft and Yandex (Schema.org, 2018). Common for all these organizations is that they have their own search engines optimized for the schema vocabulary. Search engines utilize metadata to create rich snippets presenting information about products and services, it is also used for individualized relevance ranking.

The adoption of GoodRelations ontology into the schema vocabulary resulted in similar structure, both vocabularies use their own namespace. For instance, <http://purl.org/goodrelations/v1#PaymentMethod> is represented as <http://schema.org/PaymentMethod> in the schema vocabulary. Goodrelations based the e-commerce model on the assumption that it can be represented in 4 core entities, Hepp (2015) mentions them as:

1. An agent (e.g a person or an organization),
2. A promise (offer) to transfer some rights (ownership, temporary usage, a certain license, etc.) on some object or to provide some service,
3. An object (e.g a camcorder, a house, a car, etc.) or service (e.g. a haircut), and

4. An expected compensation (e.g. an amount of money), to be provided by the accepting agent and related to the object or service.

In addition, the entity location is often used to specify where an offer is available. Furthermore, Hepp (2015) present classes in the GoodRelations vocabulary:

- gr:BusinessEntity for the agent, i.e. the company or individual,
- gr:Offering for an offer to sell, repair, lease something, or to express interest in such an offer,
- gr:ProductOrService for the object or service,
- gr:PriceSpecification for the compensation, and
- gr:Location for a store or location from which the offer is available.

The schema vocabulary has similar names on classes for e-commerce modelling. It uses class names like Organization, Offer, Product, Service, PriceSpecification and Location, with many properties available in each class. Classes are extensively used to represent different models within the e-commerce domain. Schema.org is still a work in progress with continuous implementations and updates that introduce new classes and properties. One of the newer classes is Invoice (<http://schema.org/Invoice>), which represents a receipt or bill. It has 15 properties that are used for describing attributes that are required in receipts. The properties "accountId", "confirmationNumber", "minimumPaymentDue", "paymentDueDate", "paymentMethod", "paymentMethodId", "paymentStatus", "scheduledPaymentDate" and "totalPaymentDue" are meant for identifying the transaction process. Agents involved in the transaction are described in properties like "broker", "customer" and "provider". Products are defined in orders linked through the property "referenceOrder". It is possible to define several orders, each order can contain multiple products. Below is a simple example of an Invoice in RDFa:

```
<div vocab="http://schema.org/" typeof="Invoice">
  <h1 property="description">Pizza</h1>
  <div property="broker" typeof="/LocalBusiness">
    <b property="name">Pizzabakeren</b>
  </div>

  <div property="customer" typeof="Person">
    <b property="name">Ola Nordmann</b>
  </div>

  <time property="paymentDueDate">2018-03-15</time>

  <div property="totalPaymentDue" typeof="PriceSpecification">
    <span property="price">199.00</span>
    <span property="priceCurrency">NOK</span>
  </div>

  <meta itemprop="paymentStatus" content="PaymentComplete" />

  <div property="referencesOrder" typeof="Order">
    <span property="description">pizza</span>
    <time property="orderDate">2018-03-15</time>
    <span property="orderNumber">0121446</span>
    <div property="orderedItem" typeof="Product">
      <span property="name">09 DEN MARINERTE</span>
      <meta property="productId" content="09" />
    </div>
  </div>
</div>
```

2.5 Google Knowledge Graph

Google Knowledge Graph is a database of knowledge repositories about entities or things in the real world. It is described as a graph, an intelligent model that understand entities and their relationship to one another (Singhal, 2012). The important factor is relationships between entities making it possible to build a graph. In this context, structured data with markup in RDF is useful for building and adding knowledge into the graph. The knowledge graph must be available for online search so that applications can use structured data. It is also important that generated structured data can be properly added into the knowledge graph. A major advantage is that the knowledge graph can provide answers to different queries in the same sequence, based on the relationships in data. Singhal (2012) states that one of the main purposes is to improve Google search. The knowledge graph enhance search in three ways, first one is finding the right thing by using entities. Secondly, it will present relevant content around the topic with key facts from entities. Lastly, the search will reveal other relations for an entity, making it possible to discover deeper and broader knowledge.

Uyar & Aliyu (2015) investigate three main aspects of semantic search engines, they used Google Knowledge Graph and Satori from Microsoft Bing. In their study they looked at what kinds of entity types do they cover, how common is the support for entity list searches and what kind of natural language queries do they support. The method applied was to investigate entity types based on a test set that covered general and specific entity types. Entity types were randomly selected from Freebase, a knowledge graph with approximately 2000 entity types from 76 different domains (Uyar & Aliyu, 2015, p. 202). At the time, Freebase had around 44 million of entity instances for different topics. "Since there is no way of testing the availability of entity types directly, we retrieved ten instances of each entity type from Freebase and tested their availability. If one of these instances exist on the semantic search engines as an entity, we assume that the entity type is indexed. If one

of the instances exist on the web but is not recognized by the search engines, we assume that the entity type is not indexed. To check the availability of an instance, we submit the name of the entity as the query to the search engines” (Uyar & Aliyu, 2015, p. 202). The random selection process resulted in 100 entity types, they used English interfaces for the search engines.

The results from entity type search showed that Google Knowledge Graph indexed 60 entity types and Satori managed 66 entity types. There was a total of 100 entity type in this test scenario, 25 entity types were not indexed. The range of entity types showed that both search engines covered similar and popular entity types. The searches did also pick up unindexed entity types such as ”Degree”, ”Infectious disease”, ”Olympic games” and ”File format”. These types are common and almost expected to be indexed. However, this shows that Google Knowledge Graph search and Satori are not indexing all possible entity types. Uyar & Aliyu (2015) claims that this is because of limitations in automated extraction algorithms. In particular, there is a lack of applying entity types and updating them.

Entity list searches was tested for 51 entity types that were indexed by both Google Knowledge Graph search and Satori. An entity list is the result of a query where the most relevant entities are gathered. For search engines like Google and others, it is a strategic tool for catching the attention of users without redirecting them to other websites. Results are presented in a carousel where you can move left and right, which is an appropriate interface for personal computers and smart phones. The results from entity list searches showed that Google Knowledge Graph supported ten entity types out of 51 (Uyar & Aliyu, 2015, p. 206). Satori supported seven entity types, which indicates that both search engines have a small representation of entity types for entity list search. The types found were common such as products, services, persons, organizations, attractions and places.

The last research question was about investigating natural language query interfaces of semantic search engines. Uyar & Aliyu (2015) used a data

set about US geography with 877 entries. They made queries with various complexity to see if Google Knowledge Graph search and Satori were able to correctly index entity types. This was done manually by comparing search results with test data. Results from comparison were categorized as simple queries, moderate queries, complex queries, and more complex queries. Simple queries had unambiguous intent and targeted a single entity. Moderate queries had ambiguous intent and conditional statements, such as count, max/min and transitive relations. Complex queries had two conditional statements combined with compound or nested grammatical structures. More complex queries had more than two conditional statements, they only applied three conditional statements during tests. According to the results, Google Knowledge Graph search managed to correctly answer 60 percent of simple queries (Uyar & Aliyu, 2015, p. 208). However, it did not correctly answer any of the moderate, complex, and more complex queries. Uyar & Aliyu (2015) point out three main reasons for failure:

1. Complexity of queries. Neither search engine seemed to implement advanced natural language processing techniques to parse the grammatically complex queries.
2. Unsupported terms. Search engines seem to support a limited set of terms in queries.
3. Statistical queries. Some queries involve calculation of basic statistical functions such as counting, max/min calculation and averaging. Some queries add more complexity by requiring conditional counting, conditional max/min calculation and averaging. Currently these two semantic search engines provide very limited support for statistical queries.

2.6 Data visualization

Visualization of semantic linked data increase readability and knowledge for a larger group of consumers. This way of representing data is known as infographics, a tool that enables graphical visual representations of data. One of the perks with applying infographics is to make data more readable for humans, instead of interpreting tables with rows, columns and values. Infographics are practical when presenting information in front-end designs, especially since they scale well and highlight results for further analysis.

Sgvizler2 is a modern and efficient JavaScript wrapper for visualization, it is a reboot of the original project Sgvizler by Martin G. Skjæveland. "What makes Sgvizler special is the ease with which it lets one integrate the visualization of SPARQL SELECT query result sets directly into web pages, combined with the large number of visualization types it supports and its compatibility of different origin SPARQL endpoint querying for all major modern browsers and most SPARQL endpoints" (Skjæveland, 2015, p. 362). Input parameters are grouped in different attributes, such as "data-sgvizler-endpoint" for specifying the endpoint address and "data-sgvizler-query" which holds the SPARQL query. The attribute "data-sgvizler-chart" supports a wide range of charts, while "data-sgvizler-chart-options" is used to customize the given chart model.

Skjæveland (2015) points out how Sgvizler works in practise for web development. Assuming that SPARQL queries are defined in HTML markup, each query is performed asynchronously using jQuery's ajax function. Asynchronously means that it will await an response, either XML or JSON, which is further parsed into a Google DataTable object. The DataTable with options is then drawn in a function that fills the HTML element. Google's Chart Tools contain all functions that can draw charts from DataTable objects as input, this means that new charts added in the future are automatically supported. However, it is important to ensure that the order of variables in the SELECT block are correct for SPARQL queries.

One concern that is mentioned is that JavaScript has to abide by the same origin policy. This is a security measure that prohibits a script from retrieving data from a different domain other than where the scripts lives (Skjæveland, 2015, p. 364). The solution to this problem is Cross-Origin Resource Sharing (CORS) which enables communication for external domains. However, it will only work if SPARQL endpoints are CORS enabled.

Chapter 3

Research Method

3.1 Design Science Research

In this research project with development of an artifact I have adopted the Design Science Methodology for Information and Software Engineering. It is a common and widely used methodology within the field of Information Systems and provides a well structured framework for solving problems in smaller portions. Applying design science has improved project structure, understanding of problems and revealed solutions for advancement in the research.

Hevner et al. (2004) present 7 guidelines of design science research, inherently described as problem solving process. By acquiring knowledge about a design problem, we are capable of creating solutions in the form of artifacts. The purpose of these guidelines is to "assist researcher, reviewers, editors, and readers to understand the requirements of effective design-science" (Hevner et al., 2004). Each guideline is a fraction of the problem, none of them are considered mandatory, but they should be explored for design-science research to be complete. The structure of my research project is inspired by the guidelines seen in table 3.1. They have been a good reminder of important tasks to prioritize throughout the research project.

Design-Science Research Guidelines	
Guideline	Description
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed <u>evaluation methods</u> .
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the <u>problem environment</u> .
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Table 3.1: Hevner et al. (2004)/Design Science in IS Research

3.2 Development Methodology

In relation to development methodology, I have used iterative and incremental method from agile approach towards software development. Iterative since parts of the system has been developed through repeating cycles and incremental due to following steps over time. Figure 3.1 below is a conceptual model of the iterative cycle with steps. By following these steps you ensure that important aspects are covered, since it is a cycle you will revisit steps with knowledge and experience from earlier iterations. In this sense, iterative development ensures that there is time and opportunity to make improvements.

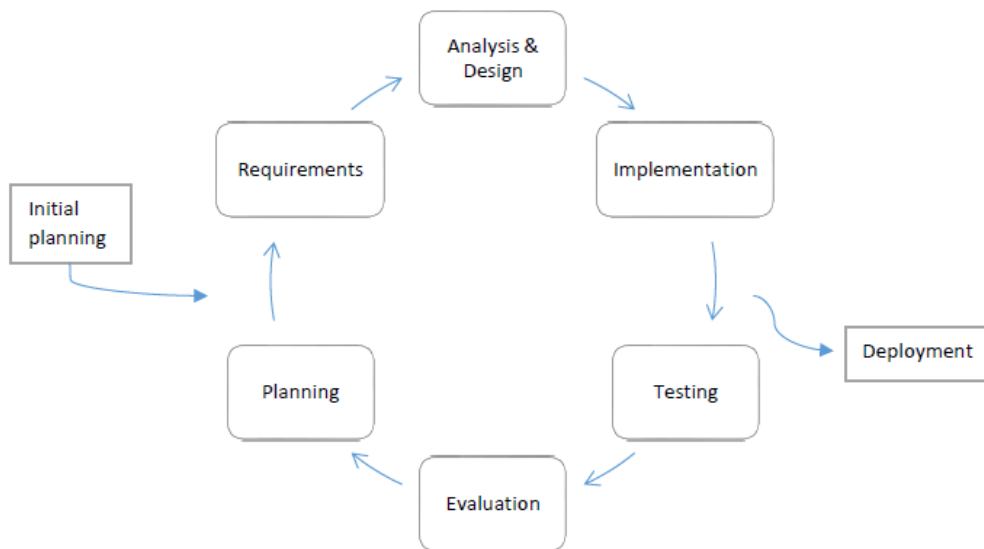


Figure 3.1: Conceptual model of the iterative cycle

Developing a system with iterative and incremental method involves implementing small parts at a time, for instance a graphical interface or module for data handling. Each requirement will run through the cycle several times until you have a result that works. The whole process of implementing requirements follows a plan that keeps an overview of components critical for making the

system work. In addition, the plan specifies time estimates for implementing requirements. However, the important factor is how many iterations you need in order to complete a requirement ready for deployment.

Scrum, Extreme Programming and Kanban are some of the prescriptive agile methodologies that are inspired by iterative and incremental development. Through the development phase of the artifact in this research project, I have not bound my development strictly to one of these methodologies. Since I have had the responsibility as the sole developer, I have borrowed some of the methods and tools for each task through its cycle.

3.3 Evaluation

The evaluation of the artefact, including data mining and semantic graph production, is based on statistics and metrics. Multiple SPARQL queries with aggregate functions have been tested on the semantic graph in order to provide data statistics. The receipts produced are compared to the model from Schema.org vocabulary (schema:Invoice), which is the reference standard. Classification searches of items in Google knowledge graph are assessed by type given and expected type. In addition, I have inspected 50 products from various online stores to find types and properties that can be reused when generating receipts from existing metadata.

3.3.1 Statistical Methods

SPARQL supports aggregate functions of data which is convenient for producing statistics. "Specifically, it provides aggregate functions COUNT, MIN, MAX, AVG, and SUM. These aggregates can be used alongside any graph pattern, computing a result for all matches for the pattern" (Allemang & Hendler, 2011, p. 101). Aggregate functions appear in the SELECT clause of queries with specified graph variable and a new variable bound to function. It is possible to group data by a valid graph variable, this is achieved with

the GROUP BY keyword. The keyword FILTER can also be used on graph variables to match specific values.

Furthermore, SPARQL supports subquery which is an additional query within a query. "Subqueries can be useful when combining limits and aggregates with other graph patterns" (Allemang & Hendler, 2011). However, subqueries of aggregates are only available in SELECT queries. Another keyword that is practical for SELECT queries with aggregate functions is UNION. "UNION combines two graph patterns, resulting in the set union of all bindings made by each pattern. Variables in each pattern take values independently (just as they do in subqueries), but the results are combined together" (Allemang & Hendler, 2011, p. 105).

There are scenarios where data sets are large and only available from the Web. In these cases it is necessary to conduct federated queries. Federate means to combine data sources, this is done via SPARQL endpoints specified as URLs. "When each data set is published via a SPARQL endpoint, SPARQL allows subqueries to be dispatched to different endpoints. The endpoint for the subquery is specified by putting the keyword SERVICE followed by a URL for the SPARQL endpoint before a graph pattern" (Allemang & Hendler, 2011, p. 110).

Drawing a trend line graph is accomplished with linear regression. "Linear regression is one of the many types of regression analysis, which models the relationship between a scalar variable y (the so called 'dependent' variable) and one or more differing and assumed independent variables x_i . A correlation between x_i and y is supposed. This kind of regression analysis is often performed when examining predictions or forecast based on observed data set of y and x . Given multiple x values the strength or grade of relation between a single variable x_j and y can be detected" (Zapilko & Mathiak, 2011, p. 120).

3.3.2 Metrics

Metrics are used for measurement, comparison and tracking performance, in relation to this research project it is relevant for information extraction and retrieval of data. Precision and recall are two acknowledged metrics for performance measurement. According to Wimalasuriya (2010), precision shows the number of correctly identified items as a proportion of total items retrieved. Recall shows the number of correctly identified items as a proportion of total number of correct items available.

$$Precision = \frac{\{Relevant\} \cap \{Retrieved\}}{\{Retrieved\}}$$

$$Recall = \frac{\{Relevant\} \cap \{Retrieved\}}{\{Relevant\}}$$

Figure 3.2: Precision and Recall equations (Wimalasuriya, 2010, p. 318)

The F-measure uses precision and recall, the equation produces a weighted average of the two metrics. "Symbol β denotes the weighting of precision versus recall. In most situations, 1 is used for β , giving equal weights for precision and recall" (Wimalasuriya, 2010, p. 318).

$$F - Measure = \frac{(\beta^2 + 1) * Relevant * Retrieved}{(\beta^2 * Precision) + Retrieved}$$

Figure 3.3: F-measure equation (Wimalasuriya, 2010, p. 318)

Chapter 4

Development

The system development phase has followed a continuous work flow progressing over time. In this chapter I will describe tools used and important aspects of the development process resulting in the final artifact.

4.1 Tools

Software development tools have been important in the long process of making the artifact. The thesis project is developed in a web environment called NodeJS, a lightweight software program built on Chrome's V8 Javascript engine (NodeJS, 2017). NodeJS builds, compiles and runs the web application. The web application holds several packages using the built in ecosystem called NodeJS package manager (npm). The ecosystem offers free access to a large registry of reusable code, some of the implemented packages will be mentioned and described in more detail later in this chapter.

During development I used Sublime Text for code writing and Bitbucket for version control. Sublime Text is my preferred text editor when working with web development, it serves the simple purpose of accessing files and editing code without any troubles. Bitbucket is a version control system available online and as software installed on your computer. Throughout

the entire development phase I used one Git repository for the project. No major problems occurred while using Bitbucket and Git, these tools have made it easier to share the work and files between my two computers. In addition, they provide valuable insight in development processes and serves as documentation. For instance, I can easily identify when project requirements were started on and completed.

Semantic tools are well represented in the web application. The knowledge graph is served and handled in Fuseki, a server for querying semantic datasets (Apache, 2017). It runs in Java and creates a localhost endpoint capable of serving RDF data over HTTP. A typical endpoint is specified as "http://localhost:3030/db", where the two last letters represent the dataset.

Data visualization is part of the front-end experience, it is available from a JavaScript library called Sgvizler2. This library can render a wide selection of graphs based on SPARQL queries. The idea behind data visualization is to present readable information and demonstrate a few examples of how knowledge graphs can be utilized.

4.2 Artifact

Based on the research questions the main idea is to develop a proof of concept web application capable of tracking receipts. Receipts are transitioning to electronic platforms and data formats, such as web services and smartphone apps delivering digital receipts to customers. Digital receipts contain detailed information about our purchases, by utilizing semantic technologies and tools such as vocabularies and classification searches it is possible to identify the things we buy. For this specific prototype, the available data is based on existing information from semi-structured digital content. Lifting semi-structured data into a semantic data model is one of the important tasks at hand.

A known challenge in this context is different data structures and how to

efficiently process and add these to the knowledge graph for receipts. The most practical case is to use established web standards such as HTML, XML or JSON. These semi-structured formats do not conform to strict models other than following the valid syntax. It is necessary to consider a lot of keywords and attributes for filtering purposes, as well as verifying that data is in the correct format and linked to the attribute it represents. Developing an algorithm that can traverse emails, sort out attributes and process data as key-value pairs is required. Once accessed, it is possible to parse these files and extract specific values which is then used in the semantic data model. RDF will be used together with relevant vocabularies, such as Schema.org to describe receipts. The process will also include Google Knowledge Graph for classification of products and services.

Once the receipts are available from the knowledge graph served by Fuseki, it is interesting to apply data visualization. First of all, the receipts need to be available and easy to retrieve. For instance, it should be possible to find a specific receipt and read off characteristics such as date, name of item(s), price and organization. Another interesting feature is to use classifications, making it possible to track how many items you buy and total cost for each category. Clearly, this feature is motivated by tracking personal receipts and cost, but there are definitely other relevant use cases for receipts stored in knowledge graphs.

4.3 Planning

The first stage was to create a plan on how to conduct phases of the development and reaching defined goals. An overview of the development process was made in a digital board called Trello (Trello, 2017). Trello is a tool available online, it is primarily a collaboration tool for development teams, but it is also applicable to one man projects. I structured my digital board with three lists named "TO DO", "IN PROGRESS" and "DONE" as shown in figure 4.1.

Each list consist of one or more cards which describes system requirements. Instead of creating many cards for specific tasks, I made them module based focusing on important system components like data mining and modelling.

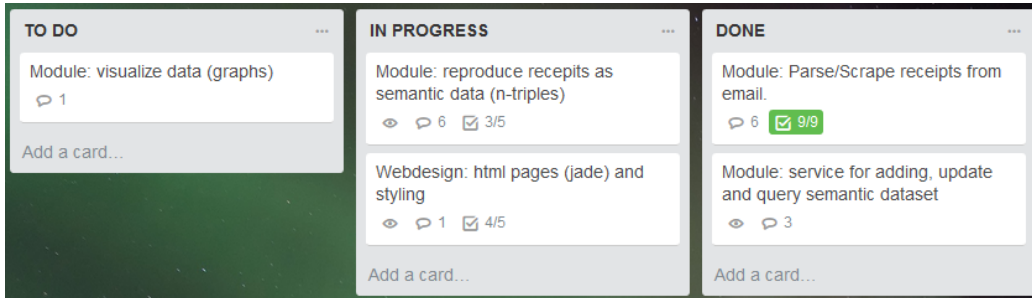


Figure 4.1: Board from Trello

Cards kept an overview of the development, the idea is to move them to the right and implement the requirements until done. For this development project, the cards stayed a long time inside the "IN PROGRESS" list, since each module contained several tasks that had to be solved. In addition, each module had to be implemented in the web application layer responsible for running modules and render views with results.

System architecture is based on a three-layered architecture that contains client, application and database layer. Client layer is the front end interface allowing users to interact with content. For this particular web application the front end experience is about exploring receipts and interact with graphs showing statistics. The application layer is defined in Node.js and includes running the web environment, utilizing modules and connect the flow of data, such as handling input and output. Database layer is responsible for storing receipts with semantic markup, also known as a knowledge graph. The web application uses a module with SPARQL support for retrieving data from the local endpoint.

4.4 Data Mining: Parsing Emails

The first card or implementation was to retrieve emails from Google Mail and parse through each one of them. Google access is given through the login webpage and stored in a session. The user has to approve that the web application is given access before the authentication process is fulfilled. Afterwards, the application will start to retrieve emails through Google Mail API. When searching through the mailbox it is possible to insert a smart filter label as parameter. I applied the following label: "label:^smartlabel_receipt", which narrowed down the search to emails classified as receipts.

Each email is identified with Multipurpose Internet Mail Extensions (MIME). It extends the email format to support text in various character sets, file attachments like audio, video, images and message bodies with multiple parts. Google Mail API return emails in JavaScript Object Notation (JSON), accessing attributes and values is achieved by using JSON-path. For instance, I could easily access attributes like sent from, date and email identification number. One of the main attributes is content, which is available as both plain text and HyperText Markup Language (HTML). I discovered that the majority of email receipts had HTML content, but the content was encoded in Base64. This is a binary to text encoding added as content attribute in the email headers, but it was possible to decode it to American Standard Code for Information Interchange (ASCII). Once decoded into a string of HTML format, you can build a Document Object Model (DOM). DOM is a tree structure where HTML tags correspond to nodes in the DOM tree hierarchy. It is a structure that can easily be traversed, such as jumping between nodes and exploring nested nodes which is necessary when parsing data. I implemented a recursive method in a module called receipt-scanner, this method traverse all the nodes in the DOM structure. A recursive method is a powerful method to solve repetitive tasks, "using recursion we can provide a solution to a problem by applying the same solution to its subproblems, an approach known as divide and conquer" (Subramaniam, 2014).

Programming the parser was a challenging task, it includes several methods and tests in order to find relevant data such as key attributes in a receipt. More specifically, the parser is searching for attributes like title of product, cost, tax, purchase method and organization. In order to find all these key attributes in receipts, I have combined traversing, pattern match and regular expressions to achieve reasonable results. The receipt-scanner module conduct multiple method calls when traversing the HTML nodes. First off, it builds a text representation of the receipt stored in a variable of type string. This is for conducting regular expression matches at a later point if the table parser is not capable of finding some of the key attributes. Secondly, the module builds an array of table nodes with text values performed by the table parser. After analysing several emails by inspecting the HTML structure, I found that key attributes describing receipts are available in table structures. Parsing tables returns data which follows a two-dimensional structure, but the order of keywords are random in each table. The reason why tables and order of keywords are different is due to different structures from organizations who send email receipts. Here is an example of how tables are interpreted by the parser seen in table 4.1 and table 4.2.

{0, 0}	{0, 1}	{0, 2}
{1, 0}	{1, 1}	{1, 2}

Table 4.1: How the parser reads a table

title/product/item	tax/subTotal/mva	total
jrollon-crj200	\$0.00	\$25.00 usd

Table 4.2: Composition of key attributes and data values

Since the keywords in the table headers appear random, it is necessary to recognize the order of specific keywords to locate the associated values. For instance, the title of a product in table 4.1 is found by linking table cell {0, 0} → {1, 0} or {title/product/item} → {jrollon-crj200}. The table is traversed

from left to right, jumping into each neighbour cell until it reaches the end and starts on a new row. As mentioned, the receipt-scanner module builds an array based on these cells and rows. By knowing the structure of the table, you can easily pair a key attribute with value through the distance between them. However, you need to make sure that each cell and row in the table is added to the array to keep the distance intact.

Considering that each email from various organizations are unique, I made a pattern file in JSON which gives instructions on how to parse tables from a given organization (see Appendix A). As for now this method is semi-automatic, meaning that the parser is not capable of reading, recognize keywords and extract values without any instructions. A better implementation is to create a more general parser capable to perceive a large set of keywords. This would require use of artificial intelligence for efficient processing with large knowledge bases, such as machine learning in natural language processing.

4.4.1 Regular Expressions

Regular expressions are extensively used in the receipt-scanner module for pattern match and extracting text. During traversing the module finds HTML tags that contain text, these tags are further processed by running customized regular expressions. Key attributes found by using these expressions are price, currency, tax and payment method. Table 4.3 below contains three regular expressions and shows how total price and currency attributes are found. In Appendix B you can see all of the expressions implemented.

First off, the expression checks if a text value matches "total" followed by a number. It will not match "subtotal" or "total before tax", since we are searching for the total amount including all costs. The price number is retrieved as text, by using a JavaScript method called "parseFloat()" it is converted to a floating point number. Lastly, we find the specified currency by matching known abbreviations for currencies.

Example	Regular Expression	Result
total: 14.9 usd (match phrase)	(?!subtotal)(?!total before tax)(total).\n.([0-9]{1,9}.[0-9]{0,2})	total: 14.9 usd
total: 14.9 usd (find number)	[0-9]+,[0-9]{1,3} [0-9]+.[0-9]{1,3}	14.9
total: 14.9 usd (find currency)	(kr usd eur gbp \$)	usd

Table 4.3: Regular Expressions for total price and currency

Combining traversing with regular expressions provide data values applicable for further modelling. The advantage of jumping between nodes and running regular expressions is processing less text and reducing the complexity of expressions. Problems that I stumbled upon when parsing email receipts was invalid formatting of HTML and strange formatting of prices. Ideally, it would have been easier to parse email receipts if they used metadata to describe content in HTML, but this is rarely the case for organizations selling products and services.

4.5 Data Modelling: Schema.org

Data retrieved from the data mining process is reproduced as semantic models. Keywords and assigned values from receipts are temporarily stored in a JSON structure as seen in figure 4.2. This made it possible to store receipts in memory and process new ones, before reproducing them as semantic structured data models.

The leap from JSON structure to semantic model is efficient and simple. Primarily, the values from each JSON receipts is inserted into a resource with markup from "schema:Invoice" (Schema.org, 2018). Resources are added to the semantic graph available through the Fuseki service. Each resource has predicates similar to the attributes defined in the JSON structure. Figure 4.3

```

var obj = {
  "id": id,
  "titles": titles,
  "total": total,
  "tax": tax,
  "currency": currency,
  "paymentMethod": paymentMethod,
  "invoiceID": invoiceID,
  "from": from,
  "email": email,
  "date": date
}

```

Figure 4.2: JSON receipt

is a test example generated in the web application that shows how receipts are represented in Turtle format. The vocabulary context is based on schema.org with mixed properties from types like Invoice, Product and Thing. There are many ways to set up a model for receipts, but I have adapted this model to the format used in email receipts and followed guidelines stated in the Schema.org vocabulary. First off I create a new instance of type Invoice, such as "invoice:1001" in figure 4.3, to represent a unique instance of the class "schema:Invoice". Each instance has a unique number attached, this id can be used to track the original email from Google. Next, type "schema:Invoice" is added along with predicates like amount, broker, email, itemListElement, paymentMethod, totalPaymentDue, priceCurrency and purchaseDate. Amount and totalPaymentDue are both associated with total cost of products, but I have used them for specific purposes. Firstly, totalPaymentDue is representing the total price in original currency for all the products included in the invoice. Amount is presenting the value of totalPaymentDue in local currency, which was norwegian krone during development. In hindsight, this could have been implemented as a personal preference in the web application. The purpose of amount is to run calculations on all the invoices at a later point in order to produce info-graphics based on costs. Naturally, it is much easier to achieve this when total price is available

```

PREFIX schema: <http://schema.org/>
PREFIX invoice: <http://schema.org/Invoice#>
PREFIX broker: <http://example/broker#>

invoice:1001
  a
    schema:Invoice ;
  schema:amount      "34"^^xsd:int ;
  schema:broker      broker:Meny ;
  schema:itemListElement [ schema:name "Banana" ] ;
  schema:itemListElement [ schema:name "Apple" ] ;
  schema:paymentMethod "visa" ;
  schema:totalPaymentDue "34" ;
  schema:priceCurrency "nok" ;
  schema:purchaseDate  "2017-11-15T00:00:01"
    ^^xsd:dateTime .

broker:Meny
  a
    "http://schema.org/GroceryStore" ,
    "http://schema.org/Corporation" ,
    "http://schema.org/Organization" ;
  schema:name "Meny" .
  schema:email "mailto:kundeservice@meny.no" ;

```

Figure 4.3: Receipt in Turtle format

as one defined currency for all invoices. Hence, I implemented a currency converter for this purpose.

Broker or organization affiliated with sale of products is added as an instance of type "schema:Organization" and "schema:Corporation". The broker in each of the receipts is not to be confused with organizations that own and manufacture products, also known as product owners. However, it is possible that a product is sold directly from the manufacturer without a middleman, meaning that broker and product owner is the same entity. Each instance of broker contains two predicates which are "schema:name" and "schema:email" retrieved from data mining. In relation to organizations, I made a simple knowledge graph in JSON-LD to demonstrate how organizations can be identified based on email addresses. Appendix C contains all of the organizations retrieved in emails when using my own Google account. The dataset is limited, however it describes reusable attributes such as types and

list of email addresses central for identification and classification. Identification is done by matching email address in broker with email addresses from the organization knowledge graph. This is done in SPARQL, it runs through all of the organizations and applies a regular expression filter based on the input email address and available addresses in "schema:email" list. Whenever the query matches an organization, it select types and add these to the broker instance as seen in figure 4.4. Seemingly, it is an option to replace broker with the matched organization, since it is the same entity with a more comprehensive model description. For future development it would be interesting to run web based queries on knowledge graphs to match specific organizations. In this regard, email addresses have served as a good filter criterion. Another challenge that occurred in the modelling process was how

```

SELECT DISTINCT ?type
WHERE
{
    ?subject ?predicate ?object .
    ?subject rdf:type ?type .
    ?subject schema:email ?email .
    FILTER regex(?email, ${inputEmail})
}

INSERT
{
    ?broker a ${type} ;
}
WHERE
{
    ?subject ?predicate ?object .
    ${invoice:receiptId} schema:broker ?broker .
}

```

Figure 4.4: Email identification

to support multiple products part of invoices. Semantic vocabularies do not support lists as in object oriented languages, such as Java and Javascript. It is only possible to specify a range of values in semantic vocabularies, for instance "rdf:type" and "schema:email" as already mentioned. The solution was to

add blank nodes in order to deal with products part of an invoice. Blank node represents a resource with an unspecified URI, also called an anonymous resource (Allemang & Hendler, 2011, p. 47). This made it possible to add unique resources representing products for every single invoice. Each product can be further described by adding type and predicates with values from vocabularies, for instance "schema:Product" accommodate several properties for this purpose.

4.6 Classification

Data manipulation between web application modules and knowledge graphs is based on SPARQL. SPARQL enables directly editing and updating of semantic graphs, a few examples has already been specified. The vision of semantic web is to make information machine readable by adding metadata. In this sense, is is also possible to reuse metadata to increase knowledge about any given resource. This is a central part of the research questions stated, whether metadata is available and how to reuse resources in knowledge graphs.

A specific case where this concept can be applied is product classification for receipts, especially since the invoices and products generated in the web application lack specific type declarations. In an attempt to explore this challenge, I added a wrapper module called "google-kgsearch", which access the Google knowledge graph through their web based Application Programming Interface (API). "The Knowledge Graph Search API lets you find entities in the Google Knowledge Graph. The API uses standard Schema.org types and is compliant with the JSON-LD specification" (Google, 2018a). By running a search on the title of a product we get JSON-LD format in response. For instance, the query string "agile principles, patterns, and practices in c#" executed in search will return the response seen in figure 4.5. From the response we can access all of the available properties, but for classification it is favourable to add type values. The types "Book" and "Thing" were returned based on the input query. All instances from Schema.org is defined with the type "Thing", this is the root or highest level of all types defined in the hierarchy. Accordingly, "Thing" is also the least precise classification of a resource and the classification module should aim for finding types with high depth relative to the hierarchy. In this sense, "Book" is a more precise type with a depth of 2 according to the hierarchy *Thing* > *CreativeWork* > *Book*. From the type definition we learn that the title used as input is the title of a book.

Schema.org currently has 597 types arranged in the hierarchy and these are used in the response elements retrieved from Google Knowledge Graph Search. In addition to the query input, it is possible to add search parameters like language code, limit, prefix and types. Type as parameter is interesting to add, but in Google Knowledge Graph it will only work if parameter type(s) input match the given type of the resource you are looking for. Therefore, I did not specify any types when using this module, it would only limit response elements or return an empty response. For instance, adding "CreativeWork" as type parameter in the search in figure 4.5 will result in an empty response, since "CreativeWork" is not part of the type values. Although "CreativeWork" is a parent class of "Book", Google knowledge graph is not reasoning with class definitions and inheritance. This means you have to match the exact type parameter to get results.

SPARQL and knowledge graphs populated with RDF triples support reasoning with class definitions and inheritance. Schema defines 597 types in the main hierarchy where each type or "class" is defined according to depth and inheritance. I downloaded the JSON-LD file of the core Schema.org vocabulary, in order to have a closer look at the hierarchy. For instance, the type "Book" is represented as seen in figure 4.6 below. Type and class definitions are described in RDFS, which builds on RDF but allows for expressing relationship between things or resources. Specifically, "rdfs:Class" and "rdfs:subClass" are used to describe class relations and inheritance. In figure 4.6 "Book" is a "rdfs:Class" and "rdfs:subClassOf schema:CreativeWork". In addition, the JSON-LD format has an attribute called "children" holding an array as value. There is one element in the array with type name "schema:Audiobook", which is a subclass of type "schema:Book". The hierarchy is as follows *Thing > CreativeWork > Book > Audiobook*, "Audiobook" with a depth of 3 is the most specific type or class available for describing books according to core Schema.org vocabulary.

```

{
  "@context": {
    "@vocab": "http://schema.org/",
    "goog": "http://schema.googleapis.com/",
    "EntitySearchResult": "goog:EntitySearchResult",
    "detailedDescription": "goog:detailedDescription",
    "resultScore": "goog:resultScore",
    "kg": "http://g.co/kg"
  },
  "@type": "ItemList",
  "itemListElement": [
    {
      "@type": "EntitySearchResult",
      "result": {
        "@id": "kg:/m/06dnh8k",
        "name": "Agile Principles, Patterns, and Practices in C#",
        "@type": [
          "Book",
          "Thing"
        ],
        "description": "Book by Micah Martin and Robert Cecil Martin"
      },
      "resultScore": 590.100525
    }
  ]
}

```

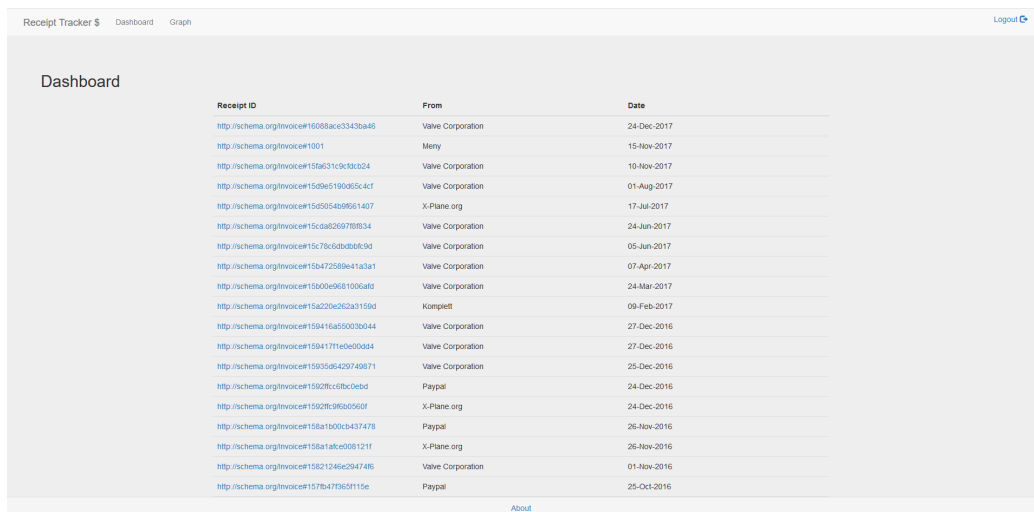
Figure 4.5: Google knowledge graph search

```
"@type": "rdfs:Class",
"rdfs:subClassOf": "schema:CreativeWork",
"description": "A book.",
"name": "Book",
"@id": "schema:Book",
"layer": "core",
"children":
[
  {
    "@type": "rdfs:Class",
    "rdfs:subClassOf": "schema:Book",
    "description": "An audiobook.",
    "name": "Audiobook",
    "@id": "schema:Audiobook",
    "layer": "bib"
  }
]
```

Figure 4.6: Book definition Schema.org

4.7 Data Visualization: Sgvizler2

Visualization of data is an important part of the front end experience and user interaction in the web application. A graphical interface makes data readable and highlights information of interest for the user. I used the tool Sgvizler2 (Rafes, 2017), a JavaScript library capable of rendering results from SPARQL select queries into charts in HTML format. The web application support two ways of interacting with receipts, first view is a "Dashboard" containing a list off all the receipts available from the knowledge graph.



The screenshot shows a web application interface titled "Receipt Tracker" with a navigation menu containing "Dashboard" and "Graph". The "Dashboard" view displays a table of receipts. The table has three columns: "Receipt ID", "From", and "Date". Each "Receipt ID" is a clickable link. The data is as follows:

Receipt ID	From	Date
http://schema.org/Invoice#16088acc3347ba46	Valve Corporation	24-Dec-2017
http://schema.org/Invoice#1601	Meny	15-Nov-2017
http://schema.org/Invoice#156631c9c5c324	Valve Corporation	10-Nov-2017
http://schema.org/Invoice#1509e5190d554cf	Valve Corporation	01-Aug-2017
http://schema.org/Invoice#156554b09614d7	X-Plane.org	17-Jul-2017
http://schema.org/Invoice#15cb820978834	Valve Corporation	24-Jun-2017
http://schema.org/Invoice#15c78c6dbdb069d	Valve Corporation	05-Jun-2017
http://schema.org/Invoice#15d472589e41a3a1	Valve Corporation	07-Apr-2017
http://schema.org/Invoice#1500e9681005af9	Valve Corporation	24-Mar-2017
http://schema.org/Invoice#15a220e262a3159d	Komplett	09-Feb-2017
http://schema.org/Invoice#150416a55003b044	Valve Corporation	27-Dec-2016
http://schema.org/Invoice#1504171e0e00004	Valve Corporation	27-Dec-2016
http://schema.org/Invoice#15035d5429740871	Valve Corporation	25-Dec-2016
http://schema.org/Invoice#1502ffc0b0c0e0d	Paypal	24-Dec-2016
http://schema.org/Invoice#1502ff0860506f	X-Plane.org	24-Dec-2016
http://schema.org/Invoice#150a1900cb437478	Paypal	26-Nov-2016
http://schema.org/Invoice#150a186e008121f	X-Plane.org	26-Nov-2016
http://schema.org/Invoice#15021246e2947486	Valve Corporation	01-Nov-2016
http://schema.org/Invoice#157b473850115e	Paypal	25-Oct-2016

Figure 4.7: Dashboard view

In figure 4.7 you see the dashboard with a table list of receipts. The table is structured in HTML and styled with Cascading Style Sheets (CSS) from Bootstrap (Otto & Thornton, 2018). Table consist of headers "Receipt ID", "From" and "Date". Table body is populated with all the receipts, each "Receipt ID" is a clickable link that opens a new view with details. Creation of list and detailed views is based on data from the knowledge graph. I implemented two methods, first method collects all the available receipts and the second for viewing details about a single receipt. Data is

selected from the knowledge graph by using SPARQL combined with views in the web application for processing and presenting data. The combination of data retrieval with SPARQL and views work well, especially since it is straightforward to write queries and process responses in JSON-LD format. It is easy to filter the results from queries, for instance the dashboard list is filtered by date showing new receipts first.

In figure 4.8 you can see a detailed view of a single receipt, it is partly inspired by type "schema:Invoice" and the structure from email receipts, although the order of properties are mixed. First attribute is a unique resource indication (URI), followed by organization name, email, date, items and total price. It is a simple and clear view consistent for all the receipts available, but it can easily be extended if new properties are added to the semantic resource.

URI	http://schema.org/Invoice#1001
From	Meny
Email	mailto:kundeservice@meny.no
Date	15-Nov-2017
Items	Apple Banana
Price	34 nok

Figure 4.8: Receipt view

On the top page you will find a fixed navigation bar, it has links to "Dashboard", "Graph" and "Logout". The "Dashboard" view is similar to how you would read and interpret emails, but "Graph" view is exclusively focusing on infographics. I implemented four types of graphs which are pie chart, bar

chart, line chart and trendline chart. Utilizing visual representation of data was one of the requirements for this web application. It is convenient to have the opportunity to read a graph, rather than interpreting a table with rows, columns and values. Graph production is based on data values like cost, date and quantity for a given type or organization. Sgvizler2 made it efficient to create graphs based on queries. The important part in these queries, was to select right data values and filter on a key attribute. In the first graph produced, the goal was to create a pie chart of things bought from various brokers. Broker is a property from schema, in this context it is used for specifying the seller as a organization. Category types from brokers are found in the query seen in figure 4.9, each slice in the pie chart represent quantity of how often a type is identified by using a count function. A significant issue in this regard is that a single broker may have multiple types. The query does not account for what type of item you bought, which would originally belong in one of the category types specified in broker. Therefore it is not a precise approach, it only provides an estimated overview of category types from brokers.


```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <http://schema.org/>
SELECT SUBSTR(?type, 19) (COUNT(?type) as ?nb)
WHERE {
  ?subject ?predicate ?object .
  ?subject a schema:Invoice .
  ?subject schema:broker ?broker .
  ?broker a ?type .
  FILTER (?type != 'http://schema.org/Organization' && ?type
    != 'http://schema.org/Corporation' && ?type != 'http://
    schema.org/CreativeWork')
} GROUP BY ?type

```

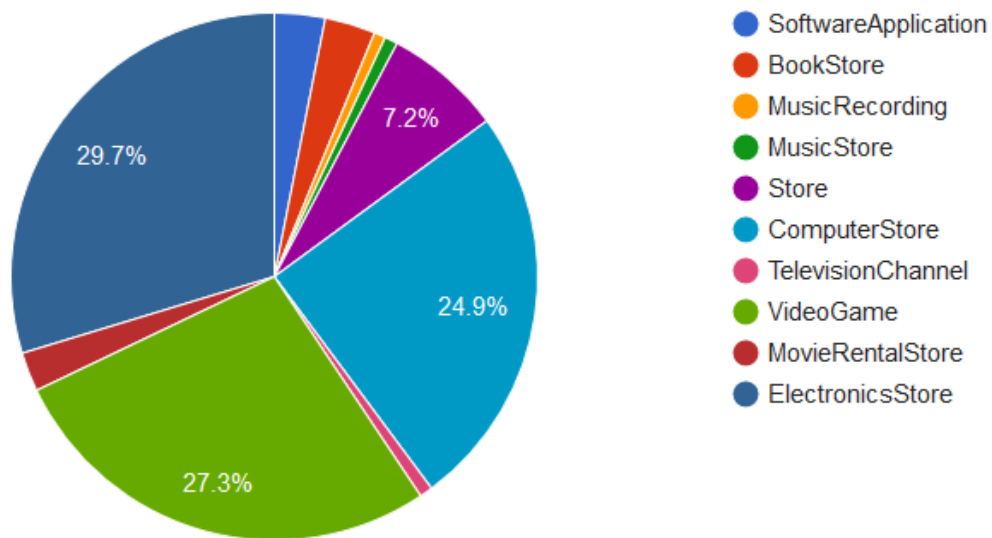


Figure 4.9: Pie chart of category types from brokers

Figure 4.10 below is similar to the first pie chart, but this chart gives an overview of item types that have been classified using Google knowledge graph.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <http://schema.org/>
SELECT ?type (COUNT(?type) as ?nb)
WHERE {
  ?subject ?predicate ?object .
  ?subject a schema:Invoice .
  ?subject schema:itemListElement ?item .
  ?item a ?type .
  FILTER (?type != schema:Thing)
} GROUP BY ?type
```

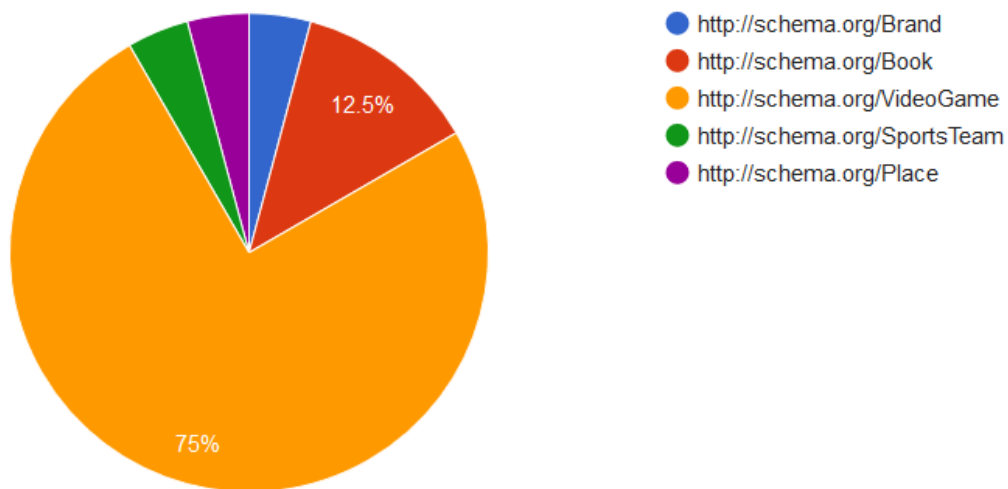


Figure 4.10: Pie chart for item types

The second graph implemented is a bar chart with organization names along the y-axis and total cost presented in rectangular bars seen in figure 4.11. It gives an overview of the current total cost for any organization based on receipts from the knowledge graph. In the third graph, I added a chart that

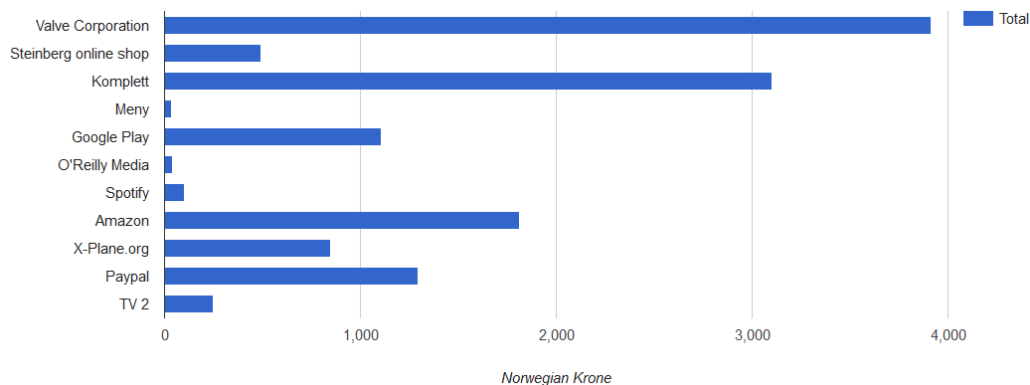


Figure 4.11: Bar chart

draws a line for 62 "schema:Invoice" resources from the knowledge graph. Each resource is placed according to date along the x-axis and total cost along the y-axis, resulting in a drawn line seen in figure 4.12.

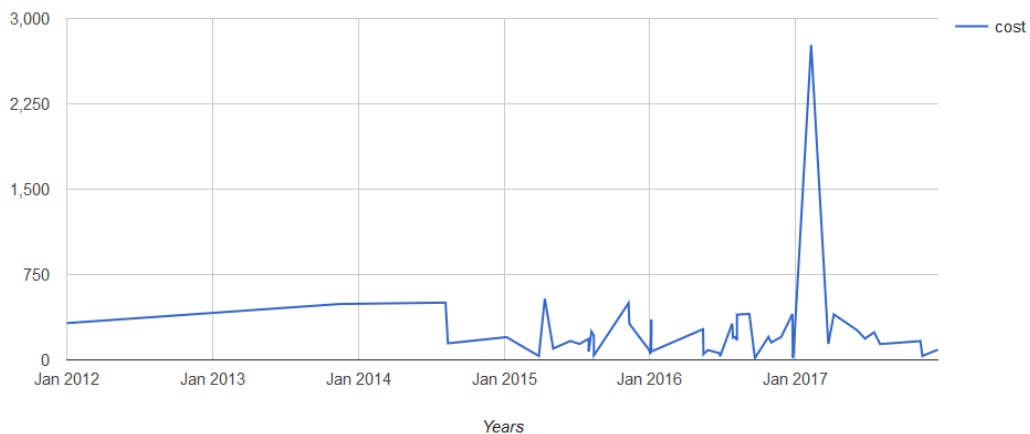


Figure 4.12: Line chart

Lastly, I added a trendline chart which is based on the mathematical principle of exponential regression. The form is e^{ax+b} , it draws the median based on the blue data points along the x-axis. The result is a slightly curved red line showing the overall trend of cost. It is intended as an indicator of whether you spend more or less money on goods and services. The curve is affected by cost and how often you buy things.

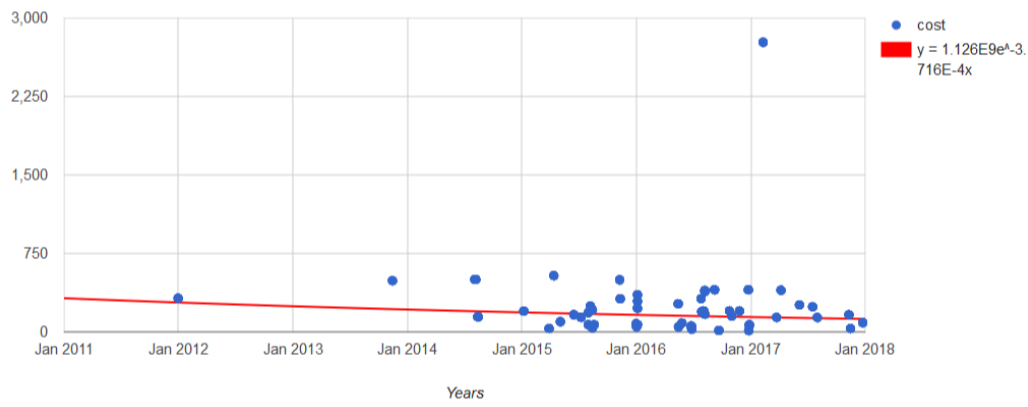


Figure 4.13: Trendline chart

Chapter 5

Analysis

In this chapter I will analyse processes like data mining, semantic modelling, classification, and finding product metadata from the web.

5.1 Data mining and semantic modelling

Data is collected from Google Mail and processed in HTML format, for development and testing I used my own Google account to retrieve emails labelled as receipts. A total of 106 emails were retrieved from the collection process. These emails are further filtered in methods were 61 out of 106 emails are qualified for semantic model production. This implies a reduction of 42.5 % using filter methods in the data mining module. Next step is production of semantic resources of type "schema:Invoice", the web application managed to successfully insert 61 resources to the endpoint "http://localhost:3030/trackReceipt". The insertion is done in two steps, first step is adding the instance of type "schema:Invoice" with properties followed by inserting items or products.

Table 5.1 gives an overview of properties belonging to all of the 61 resources. SPARQL is used to count how many properties that are present for each unique instance of type "schema:Invoice". The table shows that 6 out of 7 properties are defined for all resources, except for the property

"schema:itemListElement". This implies that 8 resources are missing the property "schema:itemListElement", which holds values of item. The underlying cause of missing properties like this one, is due to lack of data, formatting errors or shortcoming of pattern match expressions in the parser. Apart from this, it is satisfying that the other 6 properties are defined. Furthermore, two of the properties uses XML Schema Definition (XSD) for specifying data format. The property "schema:amount" has a value of "xsd:int" and "schema:purchaseDate" uses "xsd:dateTime".

Semantic resources of type "schema:Invoice".
61 resources inserted in knowledge graph.

Property	Total
schema:amount	61
schema:broker	61
schema:itemListElement	53
schema:paymentMethod	61
schema:priceCurrency	61
schema:purchaseDate	61
schema:totalPaymentDue	61

Table 5.1: Resources and properties from knowledge graph

5.2 Product classification

Items added as value to the property "schema:itemListElement" are tested for type classification in a method that runs queries through Google knowledge graph. From table 5.1 we can see that 53 items are added to resources in the knowledge graph. A count query performed in SPARQL reveals that 23 items have been given a type, this indicates that 43.4 % of the items are classified. Table 5.2 below is an overview of type classification found for item names further described in Appendix D.

The type "schema:Place" was incorrect according to assessment, item name

Type	Total
schema:Book	3
schema:Place	1
schema:VideoGame	19
schema:Brand	1
schema:SportsTeam	1

Table 5.2: Types classified for items through Google knowledge graph

was actually the title of a book, therefore "schema:Book" is expected as type. Otherwise, the method managed to classify 3 books, 1 brand, 1 SportsTeam and notable 19 video games. SportsTeam and Brand are additional types found in two of the video games. However, this also means that 30 items were not classified with a type, due to empty result from query in Google knowledge graph.

5.3 Product metadata

Exploring use of semantic markup on the web is one of the research questions. It is a relevant question for learning more about how organizations utilize semantic markup in products and services. In order to research this question further, I looked into 50 products retrieved from web stores. Metadata is gathered from 50 web pages by using Google test tool for structured data (Google, 2018b). All results from structured data gathering are available in Appendix E. There are 10 products in each of clothes, books, electronics, games and grocery categories. The class or type "Product" (<http://schema.org/Product>) from Schema.org is present in all of the collected products.

From the overview in figure 5.1 we see that types "schema:Product" and "schema:Book" are defined for products, no other types were found other than in property values. Considering that type is an appropriate classifier, it would be advantageous that metadata for products included more types and classes. In "schema:Product" it is possible to add the property "category" and

"additionalType" to specify additional types. This is actually the case for two products in the category for books. They specify type "schema:Book" before adding the property "additionalType" with value "schema:Product", which is valid markup for structuring metadata. It clearly states that this is a book and a product with available properties from these types. Another relevant property found in product is "brand", which can contribute to classification. 20 out of 50 products had the property "brand" with types "Thing" and "Brand", along with the properties name and logo.

According to findings there are 10 properties used in total from type "schema:Product". One of the important properties like "offers" contains "price" and "priceCurrency", which are also used in "schema:Invoice". There are also other relevant properties like "aggregateRating", "color", "manufacturer", "sku" (Stock Keeping Unit), "productID", "releaseDate" and "gtin" (Global Trade Item Number). In other words, the products contain metadata that can be directly reused in receipts and utilized for product classification.


```

@type: schema:Product [50], schema:Book [8]

Properties from schema:Product
* additionalProperty
* aggregateRating: { @type AggregateRating, ratingValue,
  reviewCount, bestRating, worstRating }
* color
* gtin13
* brand: { @type Brand/Thing, name, logo }
* offers: { @type Offer, @id, availability, availableAtOrFrom {
  @type Place, name}, areaServed, sku, sku13 priceCurrency,
  price, availability }
* manufacturer
* productID
* releaseDate
* sku

Properties from schema:Thing
* name
* image
* description
* url
* sameAs
* thumbnailUrl

Properties from schema:Book
* author: { @type Thing/Person, url }
* publisher: { @type Organization, name }
* genre
* dateCreated

```

Figure 5.1: Metadata product results

Chapter 6

Discussion

In this section I will discuss findings from research and development of the artefact. The research process and development have included multiple domains as reflected in the initial research questions. In short, these domains are data mining, semantic data modelling and visualization. The artefact and evaluation methods have provided interesting results which will be further discussed in this section.

6.1 Data extraction technique

The data extraction is done by recursive parsing and regular expression for finding specific attributes. These two methods are efficient and produces fair results, according to the results 61 out of 106 emails are parsed and reproduced as semantic resources. Before the parsing starts of each email, it runs a filter check on the subject header (see Appendix A). Currently there are only 19 keyword phrases, this is definitely a low number of keywords but then again there are only 106 emails which is a limited sized dataset. For a larger and more comprehensive system, it would be necessary to include more keyword phrases with language support or translating emails to English.

Traversing the semi structured HTML nodes with a recursive method is

effective, however there are various factors to consider that affect performance. These factors are typically how much input is the recursive function capable of processing, is there a proper defined base case, are there checks implemented, and not at least execution time from start to finish. Subramaniam (2014) points out the importance of using tail call optimization. This removes the concern of risking stack overflow for recursive functions with large inputs. "A tail call is a recursive call in which the last operation is a call to itself" (Subramaniam, 2014, p. 121). The recursive function implemented in the web application project uses the tail call principle. There have been no issues with stack overflows in the recursive function, but the content size in emails is also small in memory size. Usually the size of each email range from around 15 kilobytes to a few megabytes if they have attachments, but only the message body in HTML format is parsed. Regarding the base case for the implemented recursive function, it will end when there is no more nodes to loop through.

One area of improvement for the recursive method I have implemented, is to include consistent checks to prevent redundancy. In relation to regular expressions it makes sense to check if a value is set. It is not necessary to run regular expressions on remaining nodes if the value is already found. Although, it is debatable whether you need to check each node or build an comprehensive text from all the nodes, before running the regular expressions. The current approach is to check each node with text, this has proven to work well based on the results produced. By running regular expressions on each text node, you reduce the complexity of the query and each text node is quickly processed. For larger pieces of text you have to consider the structure when designing regular expressions. It will also increase the input size to be processed, which in turn will increase time consumption. A good feature with regular expressions is that they are accurate on matching keywords. It was not a problem to optimize them for keywords that are present in receipts. However, it is important to consider that there exist synonyms for keywords and that misspellings are a source of error.

In terms of time spent for the recursive function, I have measured execution time using an integrated function in Node.js, but it has not been a systematically implemented tool for analysis. The execution time for the recursive function with results from 61 emails ranged between 4 and 36 milliseconds. This shows that input size is influencing the execution time. There are also other factors such as software version and computing hardware (processor and memory) which affect execution time.

6.2 Adoption of Schema.org vocabulary

The reproduction of receipts from parsed emails to RDF triples worked well, 61 receipts or resources of type "schema:Invoice" were inserted in the Fuseki endpoint. In total, 3 of 15 properties from the type "schema:Invoice" are used in the RDF triples. The three properties are "schema:broker", "schema:paymentMethod" and "schema:totalPaymentDue". In addition there are 4 other properties from different types used for markup:

- *schema:amount* used on the types DatedMoneySpecification, InvestmentOrDeposit and LoanOrCredit. Values expected to be of one of the types: MonetaryAmount or Number.
- *schema:itemListElement* used on the type ItemList. Values expected to be one of the types: ListItem, Text, or Thing.
- *schema:priceCurrency* used on the types Offer, PriceSpecification, Reservation and Ticket. Values expected to be of type Text.
- *schema:purchaseDate* used on the types Product and Vehicle. Values expected to be one of type Date.

The property "schema:amount" was used to represent the local currency, as for now the value is a number returned from the currency convert function. There is no special property from schema that can be applied for

a currency conversion. The closest properties are "schema:amount" and "schema:currency" from type "schema:DatedMoneySpecification". The next property "schema:itemListElement" is a linked list of blank nodes that contains all the products. The products found in emails are added with this property, most of the emails only had one product. This way of modelling invoices with products differs from how it is done according to the schema vocabulary. In the type "schema:Invoice", products are added as part of an order with the property "schema:referencesOrder". This can be illustrated with an example based on one of the receipts (Figure 4.3) inserted in the knowledge graph.

```
{
  "@context": "http://schema.org/",
  "@type": "Invoice",
  "broker": {
    "@type": "GroceryStore",
    "name": "Meny",
    "email": "mailto:kundeservice@meny.no",
  },
  "customer": {
    "@type": "Person",
    "name": "Fredrik Madsen"
  },
  "paymentStatus": "http://schema.org/PaymentComplete",
  "referencesOrder": [
    {
      "@type": "Order",
      "orderDate": "2017-11-15",
      "orderNumber": "1",
      "orderedItem": {
        "@type": "Product",
        "name": "Apple",
      }
    },
    {
      "@type": "Order",
      "orderDate": "2017-11-15",
      "orderNumber": "2",
    }
  ]
}
```

```

    "orderedItem": {
      "@type": "Product",
      "name": "Banana",
    }
  ],
  "paymentMethod": "visa",
  "totalPaymentDue": {
    "@type": "PriceSpecification",
    "price": 34.00,
    "priceCurrency": "NOK",
  }
}

```

This example of a invoice shows the purchase process with orders and products added to a purchase list. The markup format is in JSON-LD, but it is possible to add this format in a query and insert it into the graph endpoint. The use of blank nodes and "schema:itemListElement" for products is similar, but it is a more general list with no references to orders.

The way orders are included in "schema:Invoice" and other types is similar to how Liu et al. (2015) envisioned information flow in e-commerce platforms. In the system proposal for e-commerce platforms, information is created, linked and updated before, during and after delivery of goods. The schema vocabulary is suitable for this kind of use, although some types and properties are missing for such a comprehensive system. In particular, logistic operations such as transport with loading, transloading and unloading to a warehouse is one of the "offline" actions that lacks information. This is also the case for e-commerce schemas, which are optimized for ordering products and payment transactions. However, applying structured data would be a good contribution in the flow of e-commerce information. This will make it easier to process data, especially when structured data formats are implemented in production systems and available for developers.

6.3 Classification of products

Classification of products with Google Knowledge graph is an applicable approach, although it only managed to classify 43.3 % of the product titles as input. It was positive that it managed to classify most of the game titles. There was one incorrect categorization of a book that was given the type "schema:Place". This misinterpretation can be caused by many reasons, such as ambiguous words, context, representative selection, filtering, etc. However, one specific critique is the lacking support of reasoning around schema types. If the Google Knowledge graph supported a SPARQL endpoint, it would be possible for developers to implement their own filtering methods in queries. This is the case for reasoning with schema types, a feature that could potentially improve filtering of products in e-commerce. The incorrect classification of the book mentioned above could have been avoided, if it was possible to include a set of types for products in the e-commerce domain.

Another challenge with classification of products, is that the types or classes used in markup of web pages are generic. By generic, it means that they rarely use subclasses of types or classes from the hierarchical structure of a vocabulary or ontology. Findings from metadata inspection of products (see Appendix E) shows that only two types, "schema:Product" and "schema:Book", from the schema.org vocabulary were in use. This provide low coverage of product types that do not contribute to better classification. In addition to specifying types, there are also other possibilities for classification in properties like "schema:category" and "schema:sameAs". The "schema:category" property can be used for specifying a category from a hierarchy, whereas "schema:sameAs" expects an URL to a web page that unambiguously indicates the identity of an item.

Stolz et al. (2014) present a semi-automatic approach for deriving classifications from existing industry standards and proprietary product category systems into product ontologies. "The tool consist of a modular architecture that builds upon three layers, namely parser, transformation process,

and serializer” (Stolz et al., 2014, p. 647). The parser reads a standard or structure of a hierarchy, which is then further processed in the internal model. Transformation process takes places in the internal model and includes creation of classes and properties describing logical rules in RDF. The internal model is finally serialized as RDF/XML. One of the good qualities in the transformation process is the use of the GenTax approach. This approach makes it possible to generate a consistent OWL ontology while preserving the taxonomic structure of the original categories from the product classification system. It makes two OWL classes, one for the taxonomic class that represents the category from the product category system. The second taxonomic class is context specific, it could for instance be within the domain of products and services. Figure 6.1 below shows an example of how GenTax would represent two items (Apple and Banana). The left hand side contains the generated categories for context product, while the right hand side have the original taxonomy classes from product classification system.

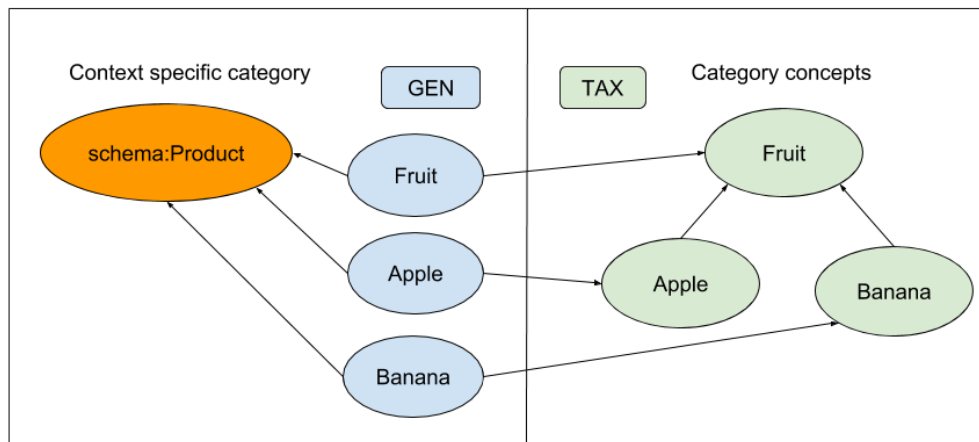


Figure 6.1: GenTax example

There is definitely a potential for the GenTax approach within product classification. First and foremost, it is applicable when parsing websites with

a product classification system where the hierarchical depth is minimum 2. It is a method that can reuse existing categories from the web, although they vary in quality. One challenge is how to merge categories without breaking the hierarchical structure, especially when dealing with more extensive ontologies with great depth.

Chapter 7

Conclusion

In this thesis I have explored how to implement and use semantic vocabularies to create receipts. I have built an artifact that is capable of parsing emails and create receipts as RDF triples. The receipts use the Invoice type from schema.org along with other types. In addition I have implemented Google Knowledge Graph search, a service capable of returning types which are used for classifying products.

7.1 Research questions

Q1. Is there sufficient metadata in web resources to categorise receipts?

There exist a considerable amount of semantic markup for products and services on the web, but according to my results and findings there is a significant improvement potential in how categories are used. First of all the types that are applied in metadata for products and services are general. There is little use of properties that are meant for classification, such as hierarchical class definition from an ontology.

Q2. What sort of categories are available in web markup for products and services?

The majority of big organizations and corporations use Schema.org with types from the vocabulary. These types are general, since they are meant to cover a wide range of things. Thus, there is a lack of more specific categories that can better distinguish resources like products and services.

Q3. What other methods can be used to enhance the category structure?

Google Knowledge graph search is a service that can provide useful categories for products and services. It managed to classify some of the product titles retrieved in the web application, but far from all.

Q4. Can we use the categories in aggregations and visualization?

Yes, it is possible to use categories in aggregations, the web application use SPARQL to retrieve data for visualizations. One particular challenge is to create queries that filter resources on available types or classes.

7.2 Future work

Recommendations for future work involves most of the domains visited and used in the development of the artefact. First off it is necessary to improve the parser in the data mining module with new capabilities. Extending the keyword lists for filtering and regular expressions is necessary along with advancing the logic for finding and recognizing HTML elements. This can be done by implementing artificial intelligence methods, such as analysing and remember the structure of data from a specific email address. The important

part is to automate the processes of reading, extracting information and save data structure as template for learning.

In the semantic module it is necessary to expand resources with more information. Receipts in the schema type "Invoice" need to include more properties with value types and literals. Organisations registered in receipts need more information, this can be done by services capable of searching through semantic metadata on the web. The currency convert function for receipts should add a resource type with properties like amount, currency and date. Classification of products and services needs more work, it can be expanded with new search methods from external services. It is preferable that these methods use knowledge graphs from the web to acquire types and classes. Further research and exploration of categories from websites is desirable, especially if it is possible to extract categories and create an ontology populated with products and services. This would generate a knowledge graph suitable for queries and classification.

Bibliography

- Allemang, Dean, James Hendler (2011). *Semantic Web for the Working Ontologist*. ISBN: 9780123859655. DOI: 10.1016/C2010-0-68657-3.
- Apache (2017). *Apache Jena Fuseki*. URL: <https://jena.apache.org/documentation/fuseki2/index.html> (visited on 03/10/2017).
- Bakar, N. S. Awang Abu (2014). ‘Using regular expressions for mining data in large software repositories’. In: *The 5th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pp. 1–6. DOI: 10.1109/ICT4M.2014.7020649.
- Berners-Lee, Tim (2006). *Linked Data*. URL: <https://www.w3.org/DesignIssues/LinkedData.html> (visited on 11/03/2018).
- Google (2018a). *Google Knowledge Graph Search API*. URL: <https://developers.google.com/knowledge-graph/> (visited on 15/03/2018).
- (2018b). *Test tool for structured data*. URL: <https://search.google.com/structured-data/testing-tool> (visited on 02/02/2018).
- gskinner (2018). *RegErr*. URL: <https://regexr.com/> (visited on 12/03/2018).
- Hepp, Martin (2015). ‘The web of data for e-commerce: Schema.org and GoodRelations for researchers and practitioners’. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9114, pp. 723–727. ISBN: 9783319198897. DOI: 10.1007/978-3-319-19890-3_66.
- Hevner, Alan R et al. (2004). ‘Design Science in Information Systems Research’. In: *MIS Quarterly* 28.1, pp. 75–105. ISSN: 02767783. DOI: 10.2307/25148625. URL: <http://dblp.uni-trier.de/rec/bibtex/journals/misq/HevnerMPR04>.
- Liu, Bingwu, Hua Hui, Juntao Li (2015). ‘The Research of Commodity E-commerce and Logistics Collaborative System Based on the Electronic Warehouse Receipts’. In: *LISS 2013*. Ed. by Runtong Zhang et al. Berlin,

- Heidelberg: Springer Berlin Heidelberg, pp. 659–666. ISBN: 978-3-642-40660-7.
- NodeJS (2017). *NodeJS Foundation*. URL: <https://nodejs.org/en/> (visited on 03/10/2017).
- Otto, Mark, Jacob Thornton (2018). *Bootstrap*. URL: <https://getbootstrap.com/> (visited on 06/01/2018).
- Rafes, Karima (2017). *Sgvizler2*. URL: <https://github.com/BorderCloud/sgvizler2> (visited on 10/01/2018).
- Schema.org (2018). *Schema.org*. URL: <https://schema.org/> (visited on 09/03/2018).
- Sheth, Amit, Krishnaprasad Thirunarayan (2012). ‘Semantics Empowered Web 3.0: Managing Enterprise, Social, Sensor, and Cloud-based Data and Services for Advanced Applications’. In: *Synthesis Lectures on Data Management* 4.6, pp. 1–175. ISSN: 2153-5418. DOI: 10.2200/S00433ED1V01Y201207DTM031.
- Singhal, Amit (2012). *Introducing the Knowledge Graph: things, not strings*. URL: <https://googleblog.blogspot.no/2012/05/introducing-knowledge-graph-things-not.html> (visited on 16/03/2018).
- Skjæveland, Martin G. (2015). ‘Sgvizler: A javascript wrapper for easy visualization of SPARQL result sets’. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7540, pp. 361–365. ISBN: 9783662466407. DOI: 10.1007/978-3-662-46641-4_27.
- Stolz, Alex et al. (2014). ‘PCS2OWL: A generic approach for deriving Web ontologies from product classification systems’. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8465 LNCS, pp. 644–658. ISBN: 9783319074429. DOI: 10.1007/978-3-319-07443-6_43.
- Subramaniam, Venkat (2014). ‘Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions’. In: *Harnessing the Power of Java 8 Lambda Expressions*, p. 171. ISSN: 0717-6163. DOI: 10.1007/s13398-014-0173-7.2.
- Trello (2017). *Trello Inc*. URL: <https://nodejs.org/en/> (visited on 10/08/2017).
- Uyar, Ahmet, Farouk Musa Aliyu (2015). ‘Evaluating search features of Google Knowledge Graph and Bing Satori’. In: *Online Information Review* 39.2, pp. 197–213. ISSN: 1468-4527. DOI: 10.1108/OIR-10-2014-0257. URL: <http://www.emeraldinsight.com/doi/10.1108/OIR-10-2014-0257>.

- W3C (2014). *RDF*. URL: <https://www.w3.org/RDF/> (visited on 10/03/2018).
- (2015). *XHTML+RDFa 1.1 - Third Edition*. URL: <https://www.w3.org/TR/xhtml-rdfa/#xhtml-rdfa-1.1-definition> (visited on 10/04/2018).
- Wimalasuriya, D. C. (2010). ‘Ontology-based information extraction: An introduction and a survey of current approaches’. In: *Journal of Information Science* 36.3, pp. 306–323. ISSN: 0165-5515. DOI: 10.1177/0165551509360123. URL: <http://jis.sagepub.com/content/early/2010/03/19/0165551509360123.abstract>.
- Zapilko, Benjamin, Brigitte Mathiak (2011). ‘Performing statistical methods on linked data’. In: *International Conference on Dublin Core and Metadata Applications*, pp. 116–125. ISSN: 19391358.

Appendix A

Subject Filter and Table Search

```
{
  "subjects": [
    "thank you for your steam purchase",
    "thank you for your in-game steam steam
      purchase",
    "thank you for your purchase",
    "thank you for your order",
    "kvittering for betalingen din",
    "amazon.com order of",
    "order of",
    "order confirmation",
    "ordrebekreftelse",
    "your subscription",
    "your order from",
    "your ebook",
    "receipt",
    "bestillingskvitteringen din",
    "bestillingsbekreftelse",
    "betalingen din til",
    "betaling til",
    "kvittering på ordre",
    "kvittering for betalingen din til"
  ]
}
```



```

{
  "patterns": [
    {
      "phrase": [{"corporation": "amazon", "text": "delivery
information", "length": true, "jump": 1},
{"corporation": "amazon", "text": "placed on", "length":
false, "jump": 1}]
    },
    {
      "phrase": [{"corporation": "komplett", "text": "pris", "
length": true, "jump": 2},
{"corporation": "komplett", "text": "beskrivelse", "length
": true, "jump": 4}]
    },
    {
      "phrase": [{"corporation": "paypal", "text": "beskrivelse
", "length": true, "jump": 4}]
    },
    {
      "phrase": [{"corporation": "steam", "text": "library page
of the base game", "length": false, "jump": 1},
{"corporation": "steam", "text": "free steam application
", "length": false, "jump": 2},
{"corporation": "steam", "text": "subtotal:", "length":
true, "jump": -2}]
    },
    {
      "phrase": [{"corporation": "spotify", "text": "items
bought:", "length": true, "jump": 1}]
    },
    {
      "phrase": [{"corporation": "x-plane.org", "text": "
download", "length": true, "jump": 2}]
    },
    {
      "phrase": [{"corporation": "the pragmatic bookstore", "
text": "item", "length": true, "jump": 4}]
    }
  ]
}

```

```
},
{
  "phrase": [{"corporation": "tv2", "text": "gratulerer med
             ditt kjc8p av", "length": false, "jump": 1}]
},
{
  "phrase": [{"corporation": "google play", "text": "vare
             ", "length": true, "jump": 2}]
}
]
```

Appendix B

Regular Expression

```
/**
 * find total cost using regexp
 *
 * @param {textNode} text input from node
 */
findTotal: function(textNode)
{
    var res2 = utils.searchRegexp("(?!subtotal)(?!total
        before tax)(\\btotal\\b)\\.\\n
        .([0-9]{1,9}\\.?[0-9]{0,2})", textNode);
    var res3 = utils.searchRegexp("(?!subtotal)(?!total
        before tax)(\\btotal\\b).*[0-9]", textNode);
    var res4 = utils.searchRegexp("(?!subtotal)(?!total
        before tax)(\\btotal\\b).*\\n.*[0-9]", textNode)
        ;
    var res5 = utils.searchRegexp("(betalt).*[0-9]",
        textNode);
    var res6 = utils.searchRegexp("(?!subtotal)(?!total
        before tax)(\\btotalt\\b).*\\n.*[0-9].*",
        textNode);
    var res7 = utils.searchRegexp("(?!subtotal)(?!total
        before tax)(\\bsum\\b).*\\n.*[0-9].*", textNode)
        ;
    if(typeof res2 !== 'undefined' && res2 !== null) {
```

```

        this.findAmount(res2);
        //console.log("res2: "+res2);
    } else if(typeof res3 !== 'undefined' && res3 !==
        null) {
        this.findAmount(res3);
        //console.log("res3: "+res3);
    } else if(typeof res4 !== 'undefined' && res4 !==
        null) {
        this.findAmount(res4);
        //console.log("res4: "+res4);
    } else if(typeof res5 !== 'undefined' && res5 !==
        null) {
        this.findAmount(res5);
        //console.log("res5: "+res5);
    } else if(typeof res6 !== 'undefined' && res6 !==
        null) {
        this.findAmount(res6);
        //console.log("res6: "+res6);
    } else if(typeof res7 !== 'undefined' && res7 !==
        null) {
        this.findAmount(res7);
    }
    },
    /**
     * find amount/number of total
     *
     * @param {res} from regexp
     */
    findAmount: function(res)
    {
        if(res !== null) {
            var resTotal = utils.searchRegexp
                ("[0-9]+,[0-9]{1,3}|[0-9]+.[0-9]{1,3}", res.
                toString().trim());
            if(resTotal !== null) {
                resTotal = resTotal.toString();
                resTotal = resTotal.replace(",",".");
                resTotal = resTotal.replace(/\s+/g, '');
            }
        }
    }
};

```

```

        var total = parseFloat(resTotal);
        receipt.total = total;
    }
    var currency = utils.searchRegex(" (kr|nok|usd|
    eur|\$)", res.toString())[0].toString();
    if(currency == null || currency.length == 0) {
        for(var i = 0; i < res.length; i++) {
            if(res[i].includes("$") || res[i].
                includes("usd")) {
                receipt.currency = "usd";
            } else if(res[i].includes("kr")) {
                receipt.currency = "kr";
            } else if(res[i].includes("nok")) {
                receipt.currency = "nok";
            } else if(res[i].includes("eur")) {
                receipt.currency = "eur";
            } else if(res[i].includes("\pounds")) {
                receipt.currency = "gbp";
            }
        }
    }
    receipt.currency = currency;
}
},

/**
 * find currency used
 *
 * @param {TextNode} text input
 */
findCurrency: function(text)
{
    if(text.includes("$") || text.includes("usd")) {
        receipt.currency = "usd";
    } else if(text.includes("\pound")) {
        receipt.currency = "gbp";
    } else if(text.includes("kr")) {
        receipt.currency = "kr";
    }
}

```

```

    } else if(text.includes("nok")) {
        receipt.currency = "nok";
    } else if(text.includes("eur")) {
        receipt.currency = "eur";
    }
},

/**
 * find value added tax using regexp
 *
 * @param {textNode} text input from node
 */
findTax: function(textNode)
{
    var res = utils.searchRegex(" (vat) (?!\\) ) .*[0-9].(
        kr)|(usd)|(eur)", textNode);
    var res2 = utils.searchRegex("(\\btax collected\\b
        )*\n.*[0-9].*", textNode);
    var res3 = utils.searchRegex("(?!total|otalt).* (
        mva).*\n.*[0-9].*", textNode);
    var res4 = utils.searchRegex("(?!subtotal|total)
        .* (\\btax\\b).*\n.*[0-9].*", textNode);
    var res5 = utils.searchRegex("(?!total)(\\bmva\\b)
        .*[0-9].*", textNode);

    if(res != null) {
        this.findTaxAmount(res);
    } else if(res2 != null) {
        this.findTaxAmount(res2);
        //console.log(res2);
    } else if(res3 != null && !res3.input.includes("
        totalt inkl.") && !res3.input.includes("id")) {
        this.findTaxAmount(res3);
        //console.log(res3);
    } else if(res4 != null) {
        this.findTaxAmount(res4);
        //console.log(res4);
    } else if(res5 != null && !res5.input.includes("

```

```

        totalt inkl.") && !res5.input.includes("id")) {
            this.findTaxAmount(res5);
            //console.log(res5);
        }
    },

    /**
     * find tax value
     *
     * @param {res} result from searchRegexp
     */
    findTaxAmount: function(res)
    {
        var taxAmount = utils.searchRegexp
            ("[0-9]+,[0-9]{1,3}|[0-9]+.[0-9]{1,3}$", res.
            toString());
        if(taxAmount == null) {
            taxAmount = utils.searchRegexp
                ("[0-9]+,[0-9]{1,3}|[0-9]+.[0-9]{1,3}", res.
                toString());
        }
        if(taxAmount != null) {
            taxAmount = taxAmount.toString();
            taxAmount = taxAmount.replace(",",".");
            var tax = parseFloat(taxAmount);
            receipt.tax = tax;
        }
    },

    /**
     * find payment method using regexp
     *
     * @param {textNode} text input from node
     */
    findPayment: function(textNode)
    {
        var res = utils.searchRegexp("(\\bvisa\\b)|(\\b
            bpaypal\\b)|(\\bmastercard\\b)", textNode);
    }
}

```

```

    if(res != null) {
        var payTitle = utils.searchRegexp("(?:visa|
            paypal|mastercard|cash|kontant)", res.
            toString());
        if(payTitle != null) {
            payTitle = payTitle.toString();
            receipt.paymentMethod = payTitle;
        }
    }
},

/**
 * find invoice id using regexp
 *
 * @param {textNode} text input from node
 */
findInvoiceID: function(textNode)
{
    var res = utils.searchRegexp("(invoice.*[0-9]|
        ordreid.*[0-9]|ordrenr.*[0-9])", textNode);
    if(res != null) {
        var invoiceId = utils.searchRegexp
            ("([0-9]{1,40})", res.toString());
        if(invoiceId != null) {
            invoiceId = parseInt(invoiceId[0].toString
                ());
            receipt.invoiceID = invoiceId;
        }
    }
}
}

```


Appendix C

Organizations JSON-LD

```
[
{
  "@context": "http://schema.org",
  "@id": "Amazon",
  "@type": ["Organization", "Corporation", "Store"],
  "email": ["mailto:auto-shipping@amazon.co.uk", "mailto:auto-shipping@amazon.co.uk", "mailto:digital-no-reply@amazon.com", "mailto:auto-confirm@amazon.co.uk"],
  "name": "Amazon"
},
{
  "@context": "http://schema.org",
  "@id": "GooglePlay",
  "@type": ["Organization", "Corporation", "CreativeWork", "BookStore", "MovieRentalStore", "SoftwareApplication", "MobileApplication"],
  "email": "mailto:googleplay-noreply@google.com",
  "name": "Google Play"
},
{
  "@context": "http://schema.org",
  "@id": "Komplett",
  "@type": ["Organization", "Corporation", "
```

```

    ElectronicsStore", "ComputerStore"],
    "email": "mailto:komplett@komplett.no",
    "name": "Komplett"
  },
  {
    "@context": "http://schema.org",
    "@id": "Meny",
    "@type": ["Organization", "Corporation", "GroceryStore"],
    "email": "mailto:kundeservice@meny.no",
    "name": "Meny"
  },
  {
    "@context": "http://schema.org",
    "@id": "Oreilly",
    "@type": ["Organization", "Corporation", "BookStore"],
    "email": "mailto:order@oreilly.com",
    "name": "O' Reilly Media"
  },
  {
    "@context": "http://schema.org",
    "@id": "PayPal",
    "@type": ["Organization", "Corporation", "PaymentMethod"],
    "email": ["mailto:service@paypal.com", "mailto:service@intl.paypal.com"],
    "name": "Paypal"
  },
  {
    "@context": "http://schema.org",
    "@id": "Spotify",
    "@type": ["Organization", "Corporation", "MusicStore"],
    "email": "mailto:no-reply@spotify.com",
    "name": "Spotify"
  },
  {

```

```

    "@context": "http://schema.org",
    "@id": "Steinberg",
    "@type": ["Organization", "Corporation", "CreativeWork", "MusicRecording", "SoftwareApplication"],
    "email": "mailto:steinberg@asknet.de",
    "name": "Steinberg online shop"
  },
  {
    "@context": "http://schema.org",
    "@id": "TV2",
    "@type": ["Organization", "Corporation", "TelevisionChannel"],
    "email": "mailto:do.not.reply@tv2.no",
    "name": "TV 2"
  },
  {
    "@context": "http://schema.org",
    "@id": "Valve",
    "@type": ["Organization", "Corporation", "ComputerStore", "VideoGame"],
    "email": "mailto:noreply@steampowered.com",
    "name": "Valve Corporation"
  },
  {
    "@context": "http://schema.org",
    "@id": "X-Plane.org",
    "@type": ["Organization", "Corporation", "VideoGame"],
    "email": "mailto:sales@x-plane.org",
    "name": "X-Plane.org"
  }
]

```

Appendix D

Classification Google knowledge graph

Item name	Schema type	Assessment
speedrunners	schema:VideoGame	correct type
agile principles, patterns, and practices in c#	schema:Book	correct type
rise of nations: extended edition	schema:VideoGame	correct type
quiplash	schema:VideoGame	correct type
playerunknown's battlegrounds	schema:VideoGame	correct type
fallout 4	schema:VideoGame	correct type
xplane 11	schema:VideoGame	correct type
blackwake	schema:VideoGame	correct type
cities: skylines	schema:VideoGame	correct type
rocket league	schema:VideoGame	correct type
squad	schema:VideoGame	correct type
learning web design: a beginner's guide to html, css, javascript, and web graphics	schema:Book	correct type
ultimate chicken horse	schema:VideoGame	correct type
african politics in comparative perspective	schema:VideoGame	correct type
sonic and all-stars racing transformed	schema:VideoGame	correct type
human-computer interaction	schema:Place	incorrect type, expected schema:Book
owlboy	schema:VideoGame	correct type
microsoft flight simulator x: steam edition	schema:VideoGame	correct type
viscera cleanup detail	schema:VideoGame	correct type
day of defeat: source	schema:VideoGame	correct type
portal 2	schema:VideoGame	correct type
sid meier's civilization v	schema:VideoGame	correct type
ark: survival evolved	schema:VideoGame	correct type

Appendix E

Metadata from products

Google test tool for structured data:

<https://search.google.com/structured-data/testing-tool>

Category: Clothes

* 2 products from XLL

Product urls:

product1

product2

Metadata:

@type: Product

properties: name, color, image, description

brand: {@type Thing, name}

offers: {@type Offer, priceCurrency, price, availability}

* 2 products from Jack & Jones

Product urls:

product1

product2

Metadata:

@type: Product

properties: sku, image, name, description

brand: {@type Brand, logo, name}

offers: {@type Offer, price, priceCurrency, itemCondition, availability}

* 2 products from Adidas

Product urls:

product1

product2

Metadata:

@type: Product

properties: name, image, description

brand: {@type Brand, name}

offers: {@type Offer, priceCurrency, price, url}

* 2 products from Ebay

Product urls:

product1

product2

Metadata:

@type: Product

properties: @id, image, name

brand: {@type Brand, name}

offers: {@type Offer, @id, itemCondition, price, availability,
priceCurrency, areaServed, availableAtOrFrom: {@type Place,
name}}

* 2 products from Boohoo

Product urls:

product1

product2

Metadata:

@type: Product

properties: @id, image, name

offers: {@type Offer, @id, url, sku, priceCurrency, price,
availability}

```

# Category: Books

* 2 products from Akademika
Product urls:
product1
product2

Metadata:
@type: Product
properties: image, name, gtin13, description

Remarks: missing use of "offers" with currency and price

* 2 products from Ark Bokhandel
Product urls:
product1
product2

Metadata:
@type: Book
additionalType: Product
properties: @id, bookFormat, schemaVersion, url, isbn,
           thumbnailURL, name, description, author: {@type: Person, name
           }

Remarks: missing use of "offers" with currency and price

* 2 products from Tandum
Product urls:
product1
product2

Metadata:
@type: Book
properties: image, name, description, author: {@type Thing, url}
offers: {@type Offer, @id, price, priceCurrency}

```


* 2 products from Ebay

Product urls:

product1

product2

Metadata:

@type: Product

properties: @id, image, name, productID,

aggregateRating: {@type AggregateRating, @id, ratingValue,
reviewCount}

offers: {@type Offer, @id, itemCondition, gtin13, price,
availability, priceCurrency, areaServed, availableAtOrFrom: {
@type Place, name}}

* 2 products from Apple iBooks

Products urls:

product1

product2

Metadata:

@type: Book

properties: name, description, image, genre, dateCreated, author
: {@type Person, name}

offers: {@type Offer, name}

publisher: {@type Organization, name}

aggregateRating: {@type AggregateRating, ratingValue,
reviewCount}

Remarks: error type value reviewCount, expected integer

```

# Category: Electronics

* 2 products from Komplet
Product urls:
product1
product2

Metadata:
@type: Product
properties: url, name, description, mpn, sku, image, ratingValue
           , reviewCount
manufacturer: {@type Organization, name}
offers: {@type Offer, availability, priceCurrency, price}

Remarks: warning missing aggregateRating with @type
         AggregateRating

* 2 products from Power
Product urls:
product1
product2

Metadata:
@type: Product
properties: name, image, url, gtin13, productID
brand: {@type Thing, name}
aggregateRating: {@type AggregateRating, ratingValue,
                 reviewCount}
offers: {@type Offer, price, priceCurrency, availability}

* 2 products from Ebay
Product urls:
product1
product2

Metadata:
@type: Product
properties: @id, image, name, mpn, model, gtin13
offers: {@type Offer, @id, itemCondition, price, availability,
        priceCurrency, areaServed, availableAtOrFrom: {@type Place,
        name}}
brand: {@type Brand, name}

```

* 2 products from AliExpress

Product urls:

product1

product2

Metadata:

@type: Product

properties: @id, name

aggregateRating: {@type AggregateRating, ratingValue,
reviewCount}

offers: {@type Offer, priceCurrency, price}

* 2 products from Kjell & Company

Product urls:

product1

product2

Metadata:

@type: Product

properties: name, description, sku, image, url

brand: {@type Brand, name}

offers: {@type Offer, price, priceCurrency, itemCondition,
availability, seller: {@type Organization, name}}

Category: Games

* 2 products from Steam

Product urls:

product1

product2

Metadata:

@type: Product

properties: image, name

aggregateRating: {@type AggregateRating, description,
reviewCount, ratingValue, bestRating, worstRating}

offers: {@type Offer, priceCurrency, price}

* 2 products from Nintendo

Product urls:

product1

product2

Metadata:

@type: Product

properties: @id, logo, description, releaseDate, sameAs, url

offer: {@type Offer, priceCurrency, availability}

isRelatedTo: {@type Thing, name}

brand: {@type Thing, name}

manufacturer: {@type Organization, name}

Remarks: schema price specification is missing in Offer

* 2 products from Gamezone

Product urls:

product1

product2

Metadata:

@type: Product

properties: @id, name, productID, description

offers: {@type Offer, @id, price, priceCurrency}

Remarks: invalid format of price

* 2 products from Ark Bokhandel

Product urls:

product1

product2

Metadata:

@type: Book

additionalType: Product

properties: @id, schemaVersion, url, isbn, thumbnailUrl, name,
description

aggregateRating: {@type: AggregateRating, ratingValue,
ratingCount, reviewCount}

Remarks: no offer type added with price and currency

* 2 products from Outland

Product urls:

product1

product2

Metadata:

@type: Product

properties: image, name, sku, description

offers: {@type Offer, price, priceCurrency}

Category: Grocery

* 2 products from Walmart

Product urls:

product1

product2

Metadata:

@type: Product

properties: name, sku, gtin13, image, description

aggregateRating: {@type AggregateRating, ratingValue, bestRating, reviewCount}

brand: {@type Thing, name}

offers: {@type Offer, priceCurrency, price, availability, itemCondition, availableDeliveryMethod, availableAtOrFrom: {@type Place, name, branchCode}}

* 2 products from Whole Foods

Product urls:

product1

product2

Metadata:

@type: Product

properties: name, description, image

Remarks: missing use of "offers" with currency and price

* 2 products from Meny

Product urls:

product1

product2

Metadata:

@type: Product

properties: name

additionalProperty: {@type PropertyValue, name}

offers: {@type Offer, priceCurrency, price}

```
* 2 products from Kolonial
Product urls:
product1
product2
Metadata:
@type: Product
properties: name, image
brand: {@type Thing, name}
offers: {@type Offer, price, priceCurrency, availability,
        itemCondition}
additionalProperty: {@type PropertyValue, name, value}

* 2 products from Mat Smart
Product urls:
product1
product2
Metadata:
@type: Product
properties: image, name

Remarks: missing use of "offers" with currency and price
```