FACULTY OF SCIENCES

Creating an Extensible Semantic Web Framework for Annotating Role Playing Game Logs

Robbe Van Herck
Student number: 01710097

Supervisor: Prof. dr. Pieter Colpaert
Counsellors: Patrick Hochstenbach, Andrei Popescu, Ruben Dedecker

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de informatica

Academic year 2021-2022

GHENT UNIVERSITY

Permission for Usage

The author gives permission to make this master's thesis available for consultation and to copy parts of this master's thesis for personal use. Every other use is subject to copyright terms, in particular with regard to the obligation to explicitly state the source when quoting results from this master's thesis.

Robbe Van Herck, 02/06/2022

Preface

This PDF version of the thesis was generated from the original HTML version at https://thesis.robbevanherck.be. Some functions such as clickable links are not available in the PDF version due to limitations of the conversion from HTML to PDF.

Acknowledgments

I want to thank my promotor, Pieter Colpaert and my counselors, Patrick Hochstenbach, Andrei Popescu and Ruben Dedecker for guiding me through the process of creating a thesis and supporting me throughout the entire process. I also want to thank Harm Delva for introducing the idea of using RPGs as a testbed for semantic web as a thesis-subject.

I want to thank Rien, Maxiem, Bastiaan, Dozens and Sammy for letting me use their storyline and characters to test the application, as well as the players in my campaigns that let me use their characters as examples in the thesis itself, no matter how chaotic the storyline may have been.

Next, I want to thank my family for giving me the opportunities to become the person I am today and for always supporting me. Of course, I also need to thank my friends who were always there to listen to my rants and helped me out when I needed it. A special thank you goes out to Zeus WPI and Jonge Helden, who let me enjoy my passions to the fullest and both contributed to my love for anything RPG-related.

Finally, I want to thank anyone that I forgot to thank, especially you, for taking the time to read the final piece of my journey through university.

Summary

When playing a role playing game (RPG) using online tools, the problem often comes up that these tools do not cooperate and that data has to be re-entered multiple times, causing inconsistent and spread-out data. This is a problem that not only exists in the world of RPG tools, but also on the world wide web as a whole. This thesis presents a proposal and proof of concept for a system that allows RPG tools to work together in a way that provides new possibilities that are not possible with the current set of tools.

More specifically, it will present a system that provides semantic tagging on the logs that players create during a campaign by performing natural language processing on them. It extracts which characters are referenced in a sentence and where using named entity recognition (NER). By solving the problems that came up along the way, it allowed experimentation with contained versions of real-world problems such as the variety of file formats.

In the end, the application was able to provide correct tagging for around 4000 pieces of text extracted from 9 different campaigns at a speed of around 1 second per sentence. The major bottleneck turned out to be automatic translation of texts, resulting in around 24% of the texts translated incorrectly. However, in the ideal vision of the project, this step is less relevant as the sentences would be inserted in the system directly instead of having to process them all before being able to use them.

Samenvatting

Tijdens het spelen van een rollenspel (RPG) met online tools komt het probleem vaak boven dat deze tools niet samenwerken en dat data vaak opnieuw ingegeven moet worden, wat leidt tot inconsistente en verspreide data. Dit is een probleem dat niet enkel bestaat in de wereld van RPG tools, maar ook in het web in zijn geheel. Deze thesis doet een voorstel en proof of concept van een systeem dat het toelaat voor RPG tools om samen te werken in een manier die nieuwe kansen brengt die niet mogelijk zijn met de huidige tools.

Meer specifiek, het stelt een systeem voor dat semantische tagging voorziet van logs die gemaakt worden door spelers tijdens een spel door natural language processing erop uit te voeren. Het extraheert welke personages werden vermeld en waar door gebruikt te maken van named entity recognition (NER). Het oplossen van de problemen liet toe om te experimenteren met versies van problemen in de echte wereld, zoals de variëteit van verschillende bestandsformaten.

Uiteindelijk was de applicatie in staat om een correcte tagging te geven van ongeveer 4000 stukken tekst van 9 verschillende verhaallijnen aan een snelheid van ongeveer 1 seconde per tekst. Het grootste probleem bleek de automatische vertaling te zijn, die resulteerde in 24% verkeerd vertaalde teksten. In de ideale visie van het project zou deze stap niet nodig zijn aangezien de teksten automatisch in het juiste formaat opgeslagen worden.

Vulgarizing Summary

Role playing games (RPGs) are a type of game where each player plays a certain character in the imaginary world of the game. All the players say what their character does and how they interact. Together, they create a storyline where the characters go on adventures, fight enemies and most of all, have fun. Often, there is one player that takes the role of the game master (GM) who controls what happens in the world, such as what monsters they encounter and what the non-player characters say and do. The GM can create challenges for the players that they need to overcome. As the game is mostly imaginary, players can come up with a virtually endless amount of ways to solve a problem. For example, if a character in the world needs eggs, the players can go look for chickens or eggs around the village, but they can also choose to lay their own eggs. Whether or not this is allowed is decided by the GM, often combined with the roll of a dice. The GM can ask the players to roll a die to see if their action succeeds, the higher the roll, the better the action succeeds.

Often, the storyline of a game continues across many sessions to form a campaign. To remember what happened in the sessions before, players often keep a log of the storyline so far. There exist many different tools to keep these logs, both offline and online. Offline tools are mostly text on paper, but some people choose to make drawings or doodles of what happened. The online tools are what this thesis revolves around. Currently, each tool has its own way of storing and processing the data and there is hardly any cooperation between tools. This thesis will create a system that helps the user make sense of their logs by analyzing them and adding annotations that indicate what player is referenced in the text. This allows them to, for example, easily search all the times a certain character is referenced in the text.

The way the system was designed also makes the data available in a format that does not just make sense to the program that created it, but it makes the data semantically meaningful. It does this by using tools and standards from the semantic web, a vision for the world wide web that makes the data on it not only accessible to the human reader, but also to a computer system that wants to understand what is on it.

After extracting data from 9 different campaigns, the system proved to be reasonably fast. It takes an average of around 1 second to fully analyze and tag a sentence, which is fast enough for the intended design where the user types a text and it gets analyzed in the background when they complete a paragraph. It proves that a system as proposed is possible and has the potential to become a whole new ecosystem of tools that all operate together to provide a better user experience.

Creating an Extensible Semantic Web Framework for Annotating Role Playing Game Logs

Robbe Van Herck , Prof. dr. Pieter Colpaert , Patrick Hochstenbach , Andrei Popescu , Ruben Dedecker

ABSTRACT

Tools that allow playing role playing games online seldomly allow exchaging data with other tools and services. This is a problem that exists on the broader web too, as websites keep users personal data in their own, separate data vaults. The semantic web provides tools that mitigate this and build a web where data is semantically meaningful so both humans and computers can understand and process it. This research presents the first steps towards a common system for RPG tools that builds on these technologies. More specifically, it presents a system that uses named entity recognition to enhance player logs and provide the first step to a new generation of tools that make the experience of playing RPGs online even more enjoyable.

1. INTRODUCTION

Players of role-playing games (RPGs) have recently had to change many of their games to an online environment. Instead of physically being at the same table in the same room, they had to use tools to simulate maps, roll dice and video chat to be able to see the other players. While this has its advantages[1], it also showed how little these tools work together. A character created in one tool is not available in another tool and has to be recreated, leading to scattered and non-synchronized data. This is a typical symptom of what is called the Web 2.0, where each application keeps its own data walled off in their own data vault. This makes interoperability between different services nearly impossible. As a response to this tendency, the *semantic web* or the Web 3.0 was created that builds on standards that allow data to be reused independently of where it came from.

In this research, a system will be presented that builds on the tools and standards of the semantic web to provide analysis and semantic enriching of the logs that players create while playing RPG games. It performs named entity recognition to extract and tag the references to characters in these logs. Besides this, the system is also designed around the interoperability of different RPG tools by allowing them to reuse the data from each step of the process.

Section 2 will show the NLP procedures that are utilized. Section 3 will show an architectural overview of the application as a whole. Section 4 will give some implementation details and finally, Section 5 will show results and draw conclusions.

2. NLP

This research mainly utilizes two tools from the field of natural language processing to achieve the goal of detecting and tagging characters in player logs. These tools are constituency parsing and named entity recognition.

2.1. Constituency Parsing

To find out what function each word has in a sentence, constituency parsing is used. This process turns a sentence into a tree where each node represents a part of the sentence. These nodes have a label indicating the function they serve in the sentence and their children represent a further subdivision of this part of the sentence. For example, the sentence “Maggie sees the castle”, becomes the constituency tree in Figure 1.

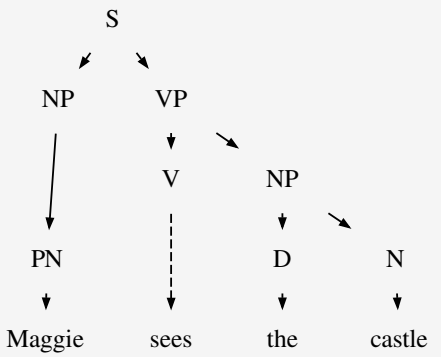

Figure 1: Constituency tree of the sentence "Maggie sees the castle". Each node in the tree represents a part of the sentence and its children represent a further subdivision. The label on the node indicates which function that part of the text serves.

This allows the selection of only the relevant parts of a sentence when processing. For example if only the verbs of a sentence are needed, only the nodes with a label of "V" can be extracted to get the required data. In a plain sentence this is more difficult to do correctly.

2.2. Named Entity Recognition

Named entities are objects, locations, persons, etc that can be identified by a name. For example in the sentence "Jeffrey sees Blub, his pet goldfish", "Jeffrey" and "Blub" are named entities, as they refer to specific characters, while "goldfish" is not, as that refers to a broad group of entities.

The process of extracting these named entities from a sentence is called named entity recognition (NER) and can be done in many different ways. This research uses constituency trees. Instead of observing every single word and matching it with the names of the entities, it only matches the words that are labeled as a noun or a noun phrase, which drastically reduces the amount of words that need to be checked.

Using this approach, it extracts references to characters that exist in an RPG campaign. For example, in the sentence from above, "Maggie sees the castle", the name of the character Maggie appears, which the algorithm would recognize as a proper noun and know it references the character named Maggie.

3. ARCHITECTURE

The basic architecture of the application is shown in Figure 2. The five steps are "extracting text from files", "extracting sentences from text", "parsing into constituency trees", "extracting characters with Solid data" and "reconstructing sentences". Each step will be explained in the next sections.

3.1. Extracting Text From Files

The first step is to process the existing log files and extract the text they contain. This is done to make sure all texts can be processed independently of their original format. In an ideal system, this step would not be necessary as the data would be stored in the correct format by the application that created the logs in the first place.

3.2. Extracting Sentences From Text

With the pieces of text known, the next step is splitting these up into sentences that can be analyzed separately. By keeping a reference from the newly split sentences to the text they originated from, it is still possible to extract context that would be lost if the sentences were stored completely separately.

3.3. Parsing Into Constituency Trees

In this step, the sentences will be parsed into constituency trees. Like in the previous steps, this does not remove any data, but adds data to the existing sentences and creates the nodes that makes up the rest of the tree. This step adds valuable metadata to each part of the sentence, as it lets the system know what the function of each part is, allowing more complex analyses.

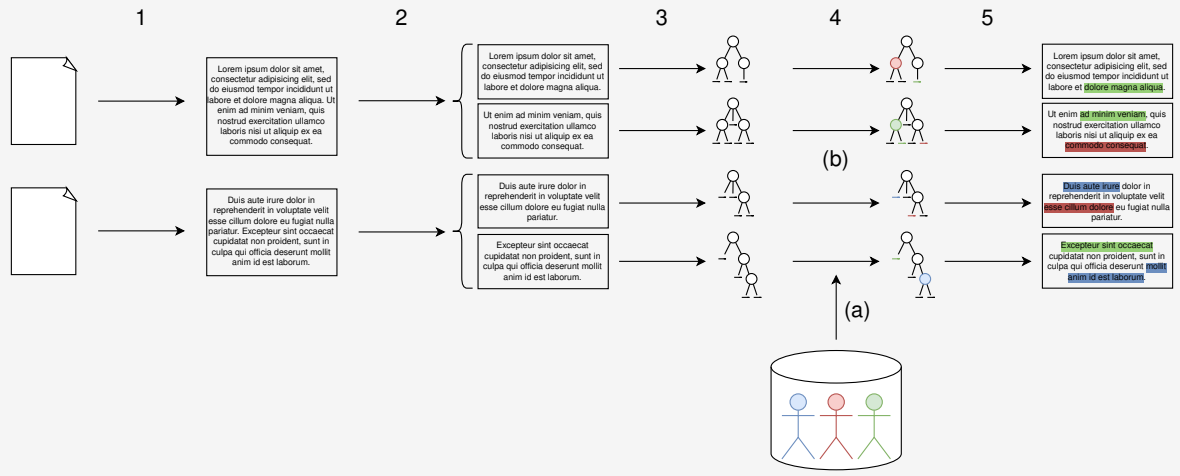

Figure 2: Overview of the design of the application. First, the raw files get transformed into pieces of text (1). Then, the pieces of text get split into sentences (2) which get parsed into constituency trees (3) and tagged with which characters they reference (4b) with data from a Solid pod (4a). Finally, the original texts are reconstructed in a semantically meaningful way (5).

3.4. Extracting Characters With Solid Data

This step consists of two parts, namely requesting the data from a Solid pod and matching this to the data that was obtained in the previous steps.

Extracting the data from the pod is done by starting at the root of the pod and recursively requesting the data it contains. This data is stored in quads, which are triples with an extra field indicating the knowledge graph or file they came from originally.

Extracting the references to characters is done by finding all the proper nouns and noun phrases from all the currently analyzed constituency trees and finding the names of all the characters. When a node that constitutes a proper noun or noun phrase has the name of a character as its value, we can assume that it references that character and add the tag to the data. It is important to not only look at proper nouns, as names that consist of two words such as “Lady Ghost” will be tagged as a noun phrase.

3.5. Reconstructing Sentences

With the constituency trees and named entity tags known, we can reconstruct sentences and texts into HTML. Using existing HTML tags and RDFa, we can create a representation that adds character metadata that people and machines can understand. To make it human-readable, the `title` attribute is used, that shows an information popup when the user hovers over the text. With this, the full name of the character and the campaign they are from can be shown. For example, the example sentence from before will show “Maggie (Pantheon Party)” when hovering over the word “Maggie”. To make it machine-readable, we use the RDFa tag `resource` with as value the URI of the character.

4. IMPLEMENTATION

This section will go over the implementation of the application. It first covers how the player logs were acquired and processed and then it will give an overview of how the application as a whole was designed. Finally, it will also show how the services stored their data in the database using existing ontologies.

4.1. Player Logs

The data that was used to test the application came from 9 different real campaigns, which allowed testing the system with the

difficulties that come with using real data. While the owners of the campaign data did give permission to use the data to test the application, permission to share or publish the data was not asked, so the exact data and the scripts to extract the texts will remain private.

Almost every campaign had a different way to store the logs. Some of them used markdown or markdown-based tools such as Obsidian[2], while others used DokuWiki[3] or a webblog hosted on their personal website. The procedure of extracting text is similar for each of the formats:

1. Strip all special characters such as links, titles and markup.
2. Extract paragraphs from the plain text.
3. Store the paragraphs.

To achieve this, a Jupyter[4] notebook was used for each campaign to query, process and store the data. This allowed the data to be inspected and verified manually without having to recalculate the data every time.

Besides the format of the original files, a common problem was the language of the logs. As most of the logs came from Dutch-speaking people, the logs were in Dutch as well and had to be translated into English before saving them. This resulted in some incorrect translations and thus some lost data. Finally, the logs were not always phrased in full, correct sentences as keeping logs can be a difficult process for players while simultaneously playing the game. The application had to be able to deal with these incorrectly phrased sentences as well.

4.2. Framework

The basis of the system is the Semantic.works[5] stack, which is a system that is built around a central triplestore and allows many services to function as reasoners on this triplestore that can read existing data and add new data extracted from it. An overview of all the services and how they interact is shown in Figure 3. Some of the services already existed and some had to be implemented specifically for this research. The services that were specifically implemented are *sentence-service*, *constituency-tree-service*, *solid-sync-service* and *named-entity-recognition-service*. All services will be briefly explained below.

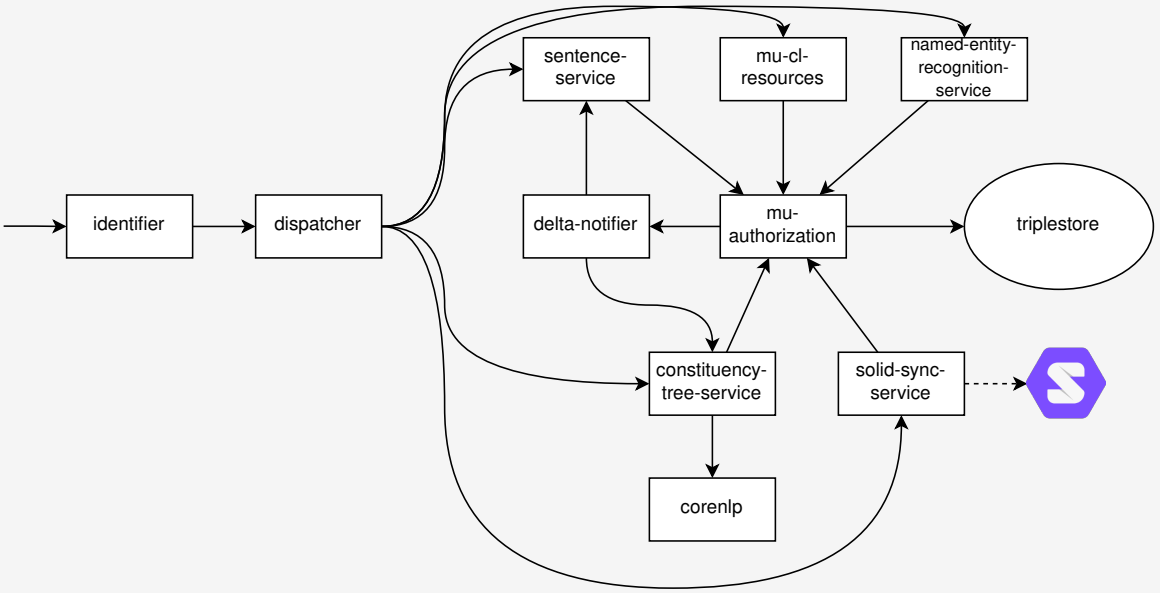

Figure 3: Overview of the application and its services. Each block represents a service and an arrow indicates an interaction from one service to the other. Requests from the frontend enter the system from the identifier.

IDENTIFIER The identifier service takes requests from the frontend and identifies to which session they belong. Mostly, this session corresponds to a browser tab the user had open.

DISPATCHER The dispatcher takes in the request forwarded from the identifier and checks which service the request should be forwarded to. This allows the application to have one shared entrypoint for all the services.

TRIPLESTORE The central data storage is the open source edition of OpenLink Virtuoso[6] database. This database allows storing triples in different graphs. It exposes a SPARQL endpoint that services can interact with to read and write data.

MU-AUTHORIZATION This service adds a layer of authorization to the database that allows certain parts of the data to be only readable or writable to certain user groups.

DELTA-NOTIFIFIER The delta-notifier notifies services of changes to the database. Every time a change is made, the service checks if any of the services are interested in it and sends them the changes directly.

MU-CL-RESOURCES To allow frontend applications to easily query the data without having to use RDF directly, a mapping between the RDF data and a JSON:API can be created with this service.

SENTENCE-SERVICE Uses a function from Pythons NLTK[7] library to split up all texts in the database into sentences. It processes new sentences immediately through the use of the delta notifier.

CONSTITUENCY-TREE-SERVICE Using a separate docker container that runs CoreNLP, this step pasres sentences into constituency trees and adds them to the database.

SOLID-SYNC-SERVICE The Community Solid Server has a feature that allows developers to perform requests under the name of a user without having to store the users password. Instead, the service stores an authentication token and uses that to perform authenticated requests. This service uses these tokens to mirror the data of one Solid pod in the database to allow easy querying for the other services.

NAMED-ENTITY-RECOGNITION-SERVICE
As both finding the names of the characters and finding all nodes of a given type can be done easily in SPARQL, this service uses a single SPARQL query to perform the NER to match the nodes to their characters.

4.3. Data Formats

To store texts, the type of `iol:Text` was used, the value of the text was stored using the `rdf:value` predicate. Sentences are stored as a `iol:Sentence` and the value stored using `rdf:value` as well. The link between texts and sentences is `dul:hasComponent`.

For constituency trees, the type of `nif:String` is added to the `iol:Sentence` that was analyzed and nodes are stored using the following predicates:

- `nif:isString`: Indicates the value of the node.
- `nif:posTag`: The OLiA part of speech tag this node has.
- `nif:subString`: Links the node to its child nodes.
- `nif:superString`: Links the node to its parent node.
- `nif:beginIndex`: The index in the parent node where the value of this node starts.
- `nif:endIndex`: The index in the parent node where the value of this node ends.

Finally, the links between nodes and characters are stored using `itsrdf:taIdentRef`.

5. RESULTS & CONCLUSION

This section will show some timing results, draw some conclusions and present the vision on how the system can evolve in the future. All timings were run on a laptop with 8GB RAM and a Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz.

The 9 campaigns consisted of 62 files, which resulted in 4138 pieces of text. Extracting sentences from these took 1 minute and resulted

in 8212 sentences, an average speed of 69 sentences per second. The constituency parsing of these took 1 hour and 11 minutes and resulted in 228055 nodes, an average speed of 1.83 sentences or 54 nodes per second. 399 sentences were not able to be parsed due to various reasons. Mirroring the data from a Solid pod took 52 seconds, after which the NER took only 8 seconds. This brings the total processing time to 1 hour and 13 minutes, which is an average of 1.06 seconds per piece of text. In a situation where these texts would be analyzed immediately after they are written, this is an acceptable wait time before getting results.

Besides the numerical results, this research also showed that it is possible to enhance the data produced by RPG players in a semantically meaningful way by using data that is itself semantically meaningful. It showed that using a shared understanding of the data allows services to cooperate in a way that is currently not available for RPG tools.

In the future, the system presented here could be expanded to become a new ecosystem for all RPG related tools. These tools could include an editor that provides real-time semantic tagging of the logs produced by the player or a system that uses VerbNet[8] matching on the constituency trees to provide event extraction or automatic summarization[9] of sessions and campaigns. In theory, every single tool that exists in the current ecosystem could be ported or recreated for this new system, providing possibilities and opportunities that are impossible with the current tools.

BIBLIOGRAPHY

[1] P. Eisenman and A. Bernstein, "Bridging the Isolation: Online Dungeons and Dragons as Group Therapy during the COVID-19 Pandemic." Available: https://www.csac-vt.org/who_we_are/csac-blog.html/article/2021/03/31/bridging-the-isolation-online-dungeons-and-dragons-as-group-therapy-during-the-covid-19-pandemic

[2] "Obsidian." Available: https://obsidian.md/

[3] "Dokuwiki [DokuWiki]." Available: https://www.dokuwiki.org/dokuwiki

[4] "Project Jupyter." Available: https://jupyter.org

[5] A. Versteden and E. Pauwels, "State-of-the-art Web Applications using Microservices and Linked Data," Zenodo, Apr. 27, 2016. doi: 10.5281/zenodo.1233427.

[6] "OpenLink Software: Virtuoso Homepage." Available: https://virtuoso.openlinksw.com/

[7] "NLTK :: Natural Language Toolkit." Available: https://www.nltk.org/

[8] M. Green, O. Hargraves, C. Bonial, J. Chen, L. Clark, and M. Palmer, "VerbNet/OntoNotes-Based Sense Annotation," in *Handbook of Linguistic Annotation*, N. Ide and J. Pustejovsky, Eds. Springer Netherlands, 2017, pp. 719–735. doi: 10.1007/978-94-024-0881-2_26.

[9] X. Han, T. Lv, Z. Hu, X. Wang, and C. Wang, "Text Summarization Using FrameNet-Based Semantic Graph Model," *Scientific Programming*, vol. 2016, pp. 1–10, Jan. 2016, doi: 10.1155/2016/5130603.

Een Uitbreidbaar Semantisch Web Framework voor de Annotatie van Rolspellogs

Robbe Van Herck , Prof. dr. Pieter Colpaert , Patrick Hochstenbach , Andrei Popescu , Ruben Dedecker

ABSTRACT

Applicaties die het toelaten om online rollenspellen te spelen laten zelden toe dat data uitgewisseld kan worden met andere applicaties. Dit is een probleem dat ook bestaat op het bredere web, waar websites de data van gebruikers in hun eigen, afgesloten data-kluizen steken. Het semantisch web voorziet middelen die dit tegengaan en bouwt een web waarbij zowel mensen als computers de data begrijpen en kunnen verwerken. Dit onderzoek toont de eerste stappen richting een gemeenschappelijk systeem voor RPG-applicaties dat op deze technologieën bouwt en de eerste stap voorziet richting een nieuwe generatie applicaties die het online spelen van RPGs nog aangenamer maken.

1. INTRODUCTIE

Spelers van rollenspellen (RPGs) hebben recent hun manier van spelen moeten aanpassen naar een online omgeving. In plaats van fysiek aan dezelfde tafel te zitten in dezelfde ruimte, moesten ze applicaties gebruiken om kaarten te simuleren, dobbelstenen te rollen en videochatten om andere spelers te kunnen zien. Hoewel dit voordelen heeft[1] toonde het ook aan hoe weinig deze applicaties samenwerken. Een personage dat gemaakt is in een applicatie is niet beschikbaar in een andere applicatie en moet opnieuw gemaakt worden, wat leidt tot versnipperde en niet-gesynchroniseerde data. Dit is een typisch symptoom van het Web 2.0, waar elke applicatie zijn eigen data afgesloten houdt in een data-kluis. Dit maakt interoperabiliteit tussen verschillende applicaties bijna onmogelijk. Als reactie op deze tendens werd het *semantisch web* of het Web 3.0 gecreëerd dat bouwt op standaarden die het toelaten om data te hergebruiken onafhankelijk van waar het kwam.

In dit onderzoek wordt een systeem voorgesteld dat bouwt op de tools en standaarden van het semantisch web om een analyse en semantische verrijking te maken van de logs die een speler maakt tijdens het spelen van een RPG spel. Het gebruikt named entity recognition (NER) om referenties naar personages te verkrijgen uit deze logs. Daarnaast is het systeem ook ontworpen rond interoperabiliteit van verschillende RPG applicaties door hen toe te laten de data van elke stap in het proces te hergebruiken.

Sectie 2 toont een overzicht van de NLP procedures die gebruikt worden. Sectie 3 zal een architecturaal overzicht geven van de applicatie in zijn geheel. Sectie 4 geeft implementatiedetails en tenslotte zal Sectie 5 resultaten tonen en conclusies trekken.

2. NLP

Dit onderzoek gebruikt voornamelijk twee tools uit de natural lanuage processing (NLP) om zijn doel van het detecteren en labelen van personages in de speler-logs te bereiken. Deze tools zijn constituency parsing en named entity recognition.

2.1. Constituency Parsing

Om te ontdekken welke functie elk woord heeft in een zin wordt constituency parsing gebruikt. Dit proces vormt een zin om in een boom waar elke knoop een deel van de zin voorstelt. Deze knopen hebben ook een label dat aanduidt welke functie ze hebben in de zin en de kinderen stellen een verdere onderverdeling van dit stuk van de zin voor. Bijvoorbeeld, de zin “Maggie sees the castle” wordt de constituency tree in Figure 4.

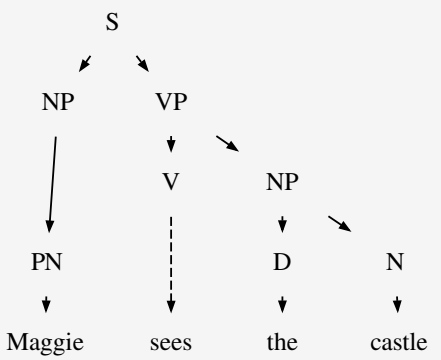

Figure 4: Constituency tree van de zin "Maggie sees the castle". Elke knoop in de boom stelt een deel van de zin voor en zijn kinderen stellen een onderverdeling van dat stuk van de zin voor. Het label van de knoop geeft aan welke functie dat stuk van de zin heeft in de volledige zin.

Dit laat toe om enkel de relevante delen van een zin te selecteren om te verwerken. Bijvoorbeeld wanneer enkel de werkwoorden van een zin nodig zijn kunnen de knopen met een label "V" (verb) geselecteerd worden om de vereiste data te verkrijgen. In een gewone zin is dit moeilijker om correct te doen.

2.2. Named Entity Recognition

Named entities zijn objecten, locaties, personen, enz. die geïdentificeerd kunnen worden met een naam. Bijvoorbeeld in de zin "Jeffrey ziet Blub, zijn goudvis" zijn "Jeffrey" en "Blub" named entities omdat ze verwijzen naar een specifiek personage, maar "goudvis" niet omdat het verwijst naar een brede groep entiteiten.

Het proces om deze named entities te vinden in een zin heet named entity recognition (NER) en kan op veel manieren gedaan worden. Dit onderzoek gebruikt constituency trees. In plaats van elk woord te bekijken en het af te toetsen aan de namen van de personages, neemt het enkel de woorden die gelabeld zijn als een eigennaam of een "noun phrase" (woordgroep die de functie van een zelfstandig naamwoord neemt), wat het aantal woorden dat bekeken moet worden drastisch verlaagt.

Bijvoorbeeld in de zin boven, "Maggie sees the castle", komt de naam van het personage Maggie voor, wat het algoritme herkent als een eigennaam en weet dat het woord verwijst naar het personage met de naam Maggie.

3. ARCHITECTUUR

De architectuur van de applicatie is te zien in Figure 5. De vijf stappen zijn "tekst verkrijgen uit bestanden", "zinnen verkrijgen uit tekst", "constituency parsing", "personages verkrijgen met Solid data" en "zinnen reconstrueren". Elke stap wordt uitgelegd in de volgende subsecties.

3.1. Tekst Verkrijgen Uit Bestanden

De eerste stap is het verwerken van de bestaande logbestanden en het verkrijgen van de tekst die ze bevatten. Dit is gedaan om zeker te zijn dat alle teksten verwerkt kunnen worden, onafhankelijk van in welk formaat ze origineel opgeslagen waren. In een ideaal systeem is deze stap niet noodzakelijk omdat de data in het juiste formaat opgeslagen wordt door de applicatie de logs creëert in de eerste plaats.

3.2. Zinnen Verkrijgen Uit Tekst

Met de tekst bekend is de volgende stap het splitsen van deze teksten in zinnen die apart verwerkt kunnen worden. Door een referentie te houden van de nieuwe gesplitste zinnen naar de tekst waar ze vandaan komen is het nog steeds mogelijk om context te verkrijgen die verloren zou gaan als de zinnen volledig apart opgeslagen werden.

3.3. Ontleden In Constituency Trees

In deze stap worden de zinnen ontleed in constituency trees. Net zoals in de vorige stap verwijdert dit geen data maar voegt het enkel data toe aan de bestaande zinnen en creëert het de knopen die de rest van de boom omvatten. Deze stap voegt kostbare metadata toe aan elk stuk van de zin, aangezien het het systeem laat weten welke functie elk stuk heeft, wat complexere analyses toelaat.

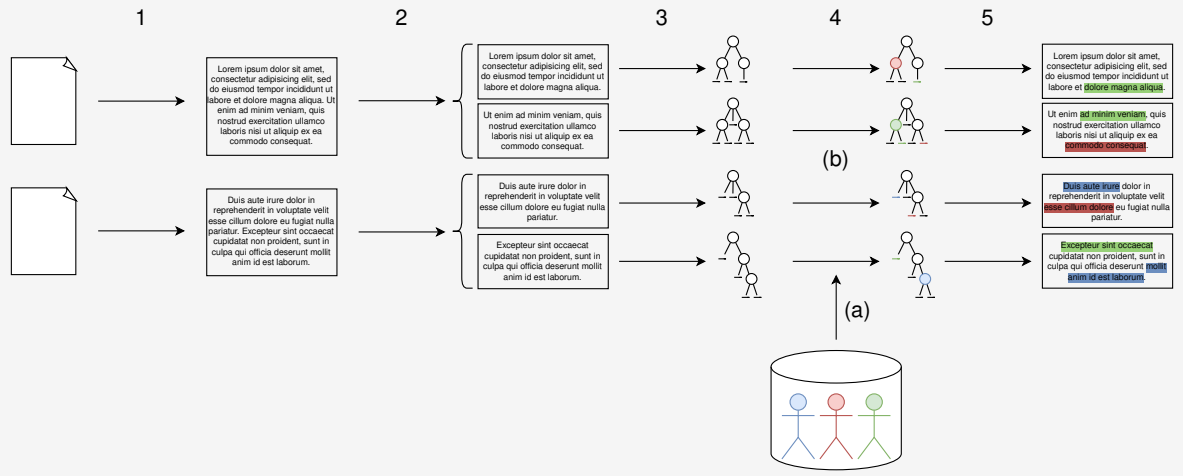

Figure 5: Overzicht van het ontwerp van de applicatie. Eerst wordt de ruwe data omgezet naar stukken tekst (1). Daarna worden de stukken tekst opgesplitst in zinnen (2) die ontleed worden in constituency trees (3) en getagd met welke peronages ze naar verwijzen (4b) met data uit een Solid pod (4a). Tot slot worden de originele teksten gereconstrueerd in een semantisch betekenisvolle manier (5).

3.4. Personages Verkrijgen Met Solid Data

Deze stap bestaat uit twee delen, namelijk het opvragen van de data van een Solid pod en dit aftoetsen aan de data die verzameld is in de vorige stappen.

Het verkrijgen van de data uit de pod is gedaan door te beginnen in de bovenste container van de pod en recursief alle data die het bevat op te vragen. Deze data wordt opgeslagen in quads, wat triples zijn met een extra veld dat aangeeft van welke knowledge graph of bestand ze origineel afkomstig zijn.

Het verkrijgen van de referenties naar personages wordt gedaan door alle eigennamen en “noun phrases” op te vragen uit de bestaande constituency trees en de namen van alle personages op te zoeken. Wanneer een knoop die een eigennaam of een “noun phrase” voorstelt als waarde de naam van een personage heeft, kunnen we ervan uit gaan dat het verwijst naar dat personage en taggen we de data ermee. Het is belangrijk om niet enkel naar knopen met als label eigennamen te kijken, aangezien het kan zijn dat namen uit meerdere woorden bestaan, zoals “Lady Ghost”, wat als label “noun phrase” zou krijgen.

3.5. Zinnen Reconstrueren

Met de constituency trees en named entity labels bekend kunnen we de zinnen en teksten reconstrueren naar HTML. Door gebruik te maken van bestaande HTML tags en RDFa, kunnen we een voorstelling maken dat personage-metadata toevoegt op een manier die zowel mensen als machines kunnen verstaan. Om het leesbaar te maken voor mensen gebruiken we het `title` attribuut, dat een informatie-popup toont als een gebruiker hovert over de tekst. Hiermee kunnen we de naam van het personage en de verhaallijn waar ze uit komen tonen. Bijvoorbeeld in de voorbeeldzin van eerder, zal “Maggie (Pantheon Party)” getoond worden wanneer er over het woord “Maggie” gehoverd wordt. Om het leesbaar te maken voor machines voegen we de RDFa tag `resource` toe met als waarde de URI van het personage.

4. IMPLEMENTATIE

Deze sectie gaat over de implementatie van de applicatie. Eerst toont het hoe de spelerlogs verzameld en verwerkt waren en daarna geeft

het een overzicht van hoe de applicatie in zijn geheel was ontworpen. Tot slot toont het ook hoe de services hun data opslaan in de database door gebruik te maken van bestaande ontologieën.

4.1. Spelerlogs

De data die gebruikt werd om het systeem te testen kwam van 9 verschillende echte verhaallijnen, wat toeliet om het systeem te testen met de moeilijkheden die komen bij het gebruik van echte data. Hoewel de eigenaars van de verhaallijnen toestemming hebben gegeven om de data te gebruiken om de applicatie te testen, was er geen toestemming gevraagd om de data te publiceren, dus de data en scripts om de teksten te verkrijgen worden niet publiek gemaakt.

Bijna elke GM had een verschillende manier om hun logs bij te houden. Sommigen gebruikten markdown of een markdowgn-gebaseerde tool zoals Obsidian[2] terwijl anderen DokuWiki[3] of een eigen webblog gebruikten. De procedure om de teksten te verkrijgen is gelijkaardig voor elk formaat:

1. Verwijder alle speciale karakters zoals links, titels en opmaak.
2. Haal de paragrafen uit de tekst.
3. Sla de paragrafen op.

Om dit te doen werd voor elke verhaallijn een Jupyter[4] notebook gebruikt om de data op te vragen, te verwerken en op te slaan. Op deze manier is het mogelijk om de data handmatig te inspecteren en verifiëren zonder de data te moeten herberekenen.

Naast het formaat van de originele bestanden was een vaak voorkomend probleem dat de meeste logs in het Nederlands geschreven waren en dus eerst vertaald moesten worden voor ze konden opgeslagen worden. Dit resulteerde soms in foute vertalingen en dus verloren data. Tot slot waren de logs ook niet altijd geformuleerd in volledige, correcte zinnen aangezien het moeilijk is voor spelers om logs bij te houden tijdens een spel. De applicatie moest hier ook mee om kunnen.

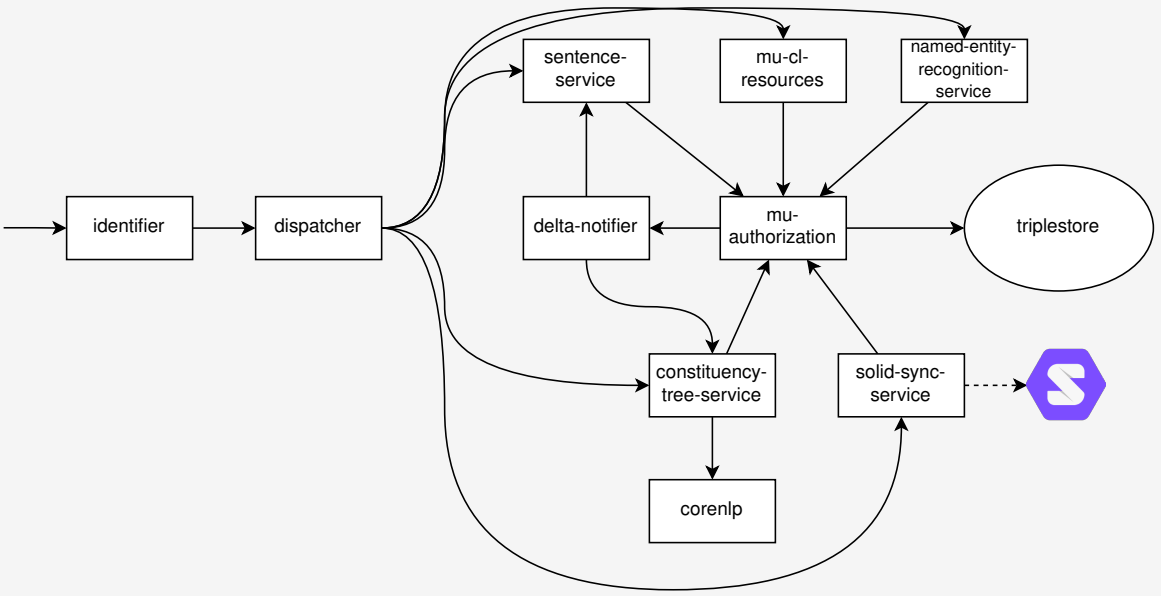

Figure 6: Overzicht van de services in de applicatie. Elk blok stelt een applicatie voor en een pijl stelt een interactie tussen twee services voor. Requests van de frontend komen binnen bij de identifier.

4.2. Framework

De basis van het systeem is Semantic.works[5], een systeem dat gebouwd is rond een centrale triplestore en toelaat voor services om te functioneren als reasoner op

deze triplestore en zo bestaande data kunnen lezen en nieuwe data toe kunnen voegen. Een overzicht van alle services en hoe ze interageren is getoond in Figure 6. Sommige servies bestonden al en anderen moesten specifiek voor dit onderzoek geïmplementeerd worden. De services die specifiek geïmplementeerd waren zijn *sentence-service*, *constituency-tree-service*, *solid-sync-service* en *named-entity-recognition-service*. Alle services worden kort uitgelegd hieronder.

IDENTIFIER De identifier service neemt requests van de frontend en identificeert tot welke sessie ze behoren. Meestal komt dit overeen met het tabblad dat de gebruiker open heeft

DISPATCHER De dispatcher neemt de request die doorgestuurd werd door de identifier en kijkt naar welke service de request doorgestuurd moet worden. Op deze manier kan een applicatie één entrypoint hebben voor alle services.

TRIPLESTORE De centrale database is de open source editie van OpenLink Virtuoso[6]. Deze database laat het toe om triples in verschillende grafen op te slaan. Het heeft een SPARQL endpoint dat services mee kunnen interageren om data te lezen en schrijven.

MU-AUTHORIZATION Deze service voegt een laag autorisatie toe aan de database die toelaat om bepaalde delen van de data enkel leesbaar of schrijfbaar te maken voor bepaalde gebruikersgroepen.

DELTA-NOTIFIFIER De delta-notifier verwittigt services wanneer er een wijziging aan de database is. Elke keer een verandering is doorgevoerd, kijkt de service of een van de services geïnteresseerd is in de wijziging en stuurt het de wijziging naar de service.

MU-CL-RESOURCES Om frontendapplicaties toe te laten om gemakkelijk de data op te vragen zonder rechtstreeks RDF te moeten gebruiken kan een omzetting van RDF data naar JSON:API en vice versa gemaakt worden met deze service.

SENTENCE-SERVICE Gebruikt een functie uit Python's NLTK[7] library om de teksten in de database op te splitsen in zinnen. Het verwerkt alle teksten wanneer ze toekomen met behulp van de delta-notifier.

CONSTITUENCY-TREE-SERVICE Met behulp van een aparte docker container die CoreNLP draait, ontleedt deze stap zinnen naar constituency trees en voegt ze toe aan de database.

SOLID-SYNC-SERVICE De Community Solid Server[8] heeft een functie die het toelaat voor ontwikkelaars om requests te sturen in naam van een gebruiker zonder het wachtwoord van de gebruiker te moeten opslaan. In plaats daarvan gebruikt de service een authenticatie-token en gebruikt deze om geauthenticeerde requests te maken. Deze service kopiëert de data van een Solid pod naar de database om het toe te laten om gemakkelijker te bevragen en verwerken.

NAMED-ENTITY-RECOGNITION-SERVICE
Aangezien zowel het vinden van de namen van de personages als het vinden van alle knopen met een gegeven typen gemakkelijk gedaan kan worden in SPARQL, gebruikt deze service een SPARQL query om de NER uit te voeren.

4.3. Data Formats

Om teksten op te slaan werd het type `iol:Text` gebruikt, de waarde van de tekst werd opgeslagen met het predicaat `rdf:value`. Zinnen werden opgeslagen als een `iol:Sentence` en de waarde met `rdf:value`. De link tussen teksten en zinnen is `dul:hasComponent`.

Voor constituency trees werd het type `nif:String` toegevoegd aan de `iol:Sentence` waar het vandaan komt en de knopen worden opgeslagen met volgende predicaten:

- `nif:isString`: Geeft de waarde van de knoop aan.
- `nif:posTag`: Het OLiA label deze knoop heeft.
- `nif:subString`: Linkt een knoop aan zijn

kinder-knopen.

- `nif:superString`: Linkt een knoop aan zijn ouder-knoop.
- `nif:beginIndex`: De index in de ouder-knoop waar de waarde van deze knoop begint.
- `nif:endIndex`: De index in de ouder-knoop waar de waarde van deze knoop eindigt.

Tot slot, de link tussen knoop en het personage waar het naar verwijst werd opgeslagen met `itsrdf:taIdentRef`.

5. RESULTAAT & CONCLUSIE

Deze sectie zal wat resultaten geven, hier conclusies uit trekken en de visie geven van hoe het systeem kan evolueren in de toekomst. Alle tijden zijn gemeten op een laptop met 8GB RAM en een Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz.

De 9 verhaallijnen bestonden uit 62 bestanden, wat resulteerde in 4138 stukken tekst. Het verkrijgen van zinnen hieruit duurde 1 minuut en resulteerde in 8212 zinnen, wat gemiddeld 69 zinnen per seconde is. Het opstellen van de constituency trees duurde 1 uur en 11 minuten en resulteerde in 228055 knopen, een gemiddelde van 1.83 zinnen of 54 knopen per seconde. Het kopiëren van de Solid data duurde 52 seconden, waarna de NER 8 seconden duurde. Dit brengt het totaal op 1 uur en 13 minuten, wat gemiddeld 1.06 seconden is per tekst. In een situatie waar deze teksten onmiddelijk verwerkt worden nadat ze geschreven worden is dit een redelijke wachttijd, zeker aangezien de berekeningen in de achtergrond kunnen gebeuren.

Naast de numerieke resultaten heeft dit onderzoek ook aangetoond dat het mogelijk is om de data geproduceerd door RPG spelers op een semantische manier te verbeteren door data te gebruiken die inherent semantische betekenis heeft. Het heeft aangetoond dat een gedeeld begrip van de data toelaat dat services samenwerken in een manier die op dit moment niet mogelijk is met bestaande RPG tools.

In de toekomst kan het systeem uitbreid worden om een nieuw ecosysteem te worden voor alle RPG gerelateerde tools. Deze tools zijn dingen als een editor die in live semantische tagging voorziet van de logs of een systeem dat VerbNet[9] gebruikt voor gebeurtenis-extractie of automatische samenvatting van sessies[10]. In theorie zou elke bestaande tool in het ecosysteem geïntegreerd kunnen worden, wat mogelijkheden en kansen met zich meebrengt die op dit moment onmogelijk zijn met de bestaande tools.

BIBLIOGRAFIE

[1] P. Eisenman and A. Bernstein, “Bridging the Isolation: Online Dungeons and Dragons as Group Therapy during the COVID-19 Pandemic.” Available: https://www.csac-vt.org/who_we_are/csac-blog.html/article/2021/03/31/bridging-the-isolation-online-dungeons-and-dragons-as-group-therapy-during-the-covid-19-pandemic

[2] “Obsidian.” Available: https://obsidian.md/

[3] “Dokuwiki [DokuWiki].” Available: https://www.dokuwiki.org/dokuwiki

[4] “Project Jupyter.” Available: https://jupyter.org

[5] A. Versteden and E. Pauwels, “State-of-the-art Web Applications using Microservices and Linked Data,” Zenodo, Apr. 27, 2016. doi: 10.5281/zenodo.1233427.

[6] “OpenLink Software: Virtuoso Homepage.” Available: https://virtuoso.openlinksw.com/

[7] “NLTK :: Natural Language Toolkit.” Available: https://www.nltk.org/

[8] “CommunitySolidServer/CommunitySolidServe An Open and Modular Implementation of the Solid Specifications.” Available: https://github.com/CommunitySolidServer/CommunitySolidServer

[9] M. Green, O. Hargraves, C. Bonial, J. Chen, L. Clark, and M. Palmer, “VerbNet/OntoNotes-Based Sense Annotation,” in *Handbook of Linguistic Annotation*, N. Ide and J. Pustejovsky, Eds. Springer Netherlands, 2017, pp. 719–735. doi: 10.1007/978-94-024-0881-2_26.

[10] X. Han, T. Lv, Z. Hu, X. Wang, and C. Wang, “Text Summarization Using FrameNet-Based Semantic Graph Model,” *Scientific Programming*, vol. 2016, pp. 1–10, Jan. 2016, doi: 10.1155/2016/5130603.

Table of Contents

List of Figures

List of Listings

List of Acronyms

RPG	Role-Playing Game
NLP	Natural Language Processing
JSON	JavaScript Object Notation
RDFa	RDF in Attributes
GM	Game Master
HTML	HyperText Markup Language
CSS	Cascading StyleSheet
NER	Named Entity Recognition

List of Ontologies

foaf	Friend of a Friend
schema	Schema.org
ttrpg	TTRpg
rdf	RDF
iol	Information Objects lite
dul	DOLCE+DnS Ultralite
mu	mu.semtech
nif	NLP Interchange Format
olia	Ontologies of Linguistic Annotation
itsrdf	ITS 2.0 / RDF

Chapter 1: Introduction

This thesis presents the first steps to creating a system that improves the experience of playing role playing games (RPGs) in an online environment, as recently more and more campaigns are happening online. While playing online has its benefits such as being able to play independently of where players are in the real world, it also has its drawbacks. A major problem is the fact that the plethora of tools available do not work together. In this thesis, the groundwork will be laid for a system that allows many different tools to work together, independently of their exact function.

The focus in this thesis will be to provide enhancements to the process of keeping logs during a game by analyzing the text and adding relevant metadata. More specifically, it will analyze these texts and extract the names of characters that exist in the campaign and add a link to their resource to the data. Instead of inventing data formats from the ground up, this thesis builds on the work done in the field of semantic web.

The ideal future of the project is a system where there is not just this one task of analyzing the logs of the game, but a system where different tools such as character creators, dice simulators and world maps cooperate by making use of and contributing to the same dataset. This data could then be enhanced by different reasoners and services to add useful metadata, such as finding out which player has more luck on dice rolls, combining the logs of different players or, in the case of this research, annotating text with character metadata.

This thesis will not create a system where each task is a pipeline that takes input, performs a task and outputs the result, but a system where every step in the process produces useful data that can be reused by other systems. The former is shown in Figure 7, where not every system supports every file type, processing steps are done multiple times and the resulting formats are incompatible. The latter is shown in Figure 8, where the file formats are converted into a shared format, and analysis is done by services that work on this shared format, instead of on the file formats directly. This also results in more rich data exported from the service.

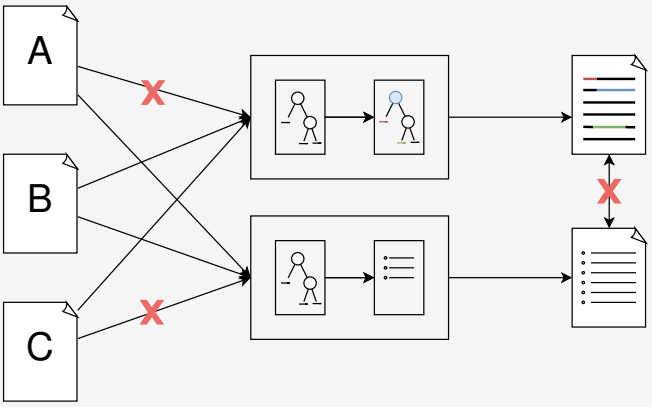

Figure 7: Schematic overview a classic system layout where each of the programs only supports a limited set of file types and produces output that is not compatible with the other formats.

Since RPGs are intrinsically free-form and improvisational, using them as a testbed poses both interesting challenges and provides unique opportunities. Two examples of this are described below.

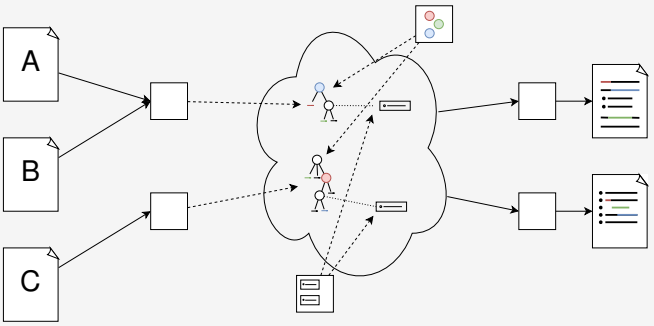

Figure 8: Schematic overview of a system with shared data where the files first get transformed into a shared format that the services can enhance by adding more data to it. Other services can then extract semantically meaningful data from it and produce semantically rich output formats.

The first example is the fact that data comes in many different formats, in the case of RPGs, this is because each player uses a method to keep track of their character and story line that suits them most. Some keep logs in a shared document, others use wikipedia-style web applications and some even use drawings on paper. If we want to create a system that the most amount of people can use, we also want to support as many of these formats as we can. In the real world, this problem exists too. Many documents only exist as PDF, as a Microsoft Word document or even as an image of a scanned document. Tools that want to use the data from these documents needs to support most of them as well.

The second example is the fact that within an RPG world, certain assumptions can be made to make processing the data easier. For example, in a well-documented RPG campaign, we can assume that a reference to every character, object and location that has been seen by the players exists. However, if it has not been seen by the players, that does not mean it doesn't exist as it might be that the game master (GM) has not prepared that part of the world yet. This allows us to experiment with semantic webs *open world assumption*, which states that a fact is not necessarily false if we don't know it is true. In the real world, we also assume that we will never have all the data of everything in existence, so we need to be able to deal with this incomplete data.

The rest of this thesis is structured as follows: Chapter 2 will give an overview of the literature related to the subject of this thesis. Chapter 3 provides an overview of the semantic web technologies that are used and provides more context on how they work. Chapter 4 describes the natural language processing techniques that are used. Chapter 5 explains the architectureof each part in the application. Chapter 6 explains the technical choices made when implementing the application. Chapter 7 will show and evaluate the final result as a whole and finally, Chapter 8 draws conclusions and lays out possible future work.

Chapter 2: Literature Review

1. NLP on the Semantic Web

The idea to use NLP on the semantic web is not new. Many approaches and ideas have been proposed over the years. Wilks and Brewster[1] proposed in 2009 that a to achieve a semantic web, the usage of NLP was required to extract data from the world wide web and transform it into RDF.

In Natural Language Processing for the Semantic Web[2], Maynard et al. explain the usage of NLP, how they can be used to enhance the semantic web and vice versa. They also talk about the difficulties in natural language processing, such as noisy content (the inclusion of emoticons, capitalization, …).

A recent survey by Martinez-Rodriguez from 2020[3] shows that while the evolution of information extraction is still ongoing, there has been a shift from specific, domain-limited analyses to more broad approaches. For example, where NER was often used to extract entities from a custom dataset, it is now more and more used to extract entities from larger, general datasets such as DBpedia.

In an effort to combine the plethora of NLP tools that were created, Hellmann et al.[4] presented NIF, which would serve as a shared ontology for all NLP tools to be able to cooperate and understand the data that was produced.

Some tools were also created that utilize the existing standards and ideas to provide semantically rich tagging of data. One of these is FRED[5] which is a general-purpose text annotation tool. It performs sentence analysis and stores the results in a RDF graph. This provides a lot of interesting data, however because FRED uses custom ontologies for many of their resources, the exact semantic meaning becomes unclear.

Another system is LODifier[6], which performs many NLP tasks and combines the data of all these steps in a RDF knowledge graph. A major shortcoming of LODifier that is described in the paper is that the system uses a pipelined approach, resulting in problems for things like error recovery.

Finally, the system that resembles the system proposed in this thesis most is the SlugNERDS[7]. Their approach also uses constituency parsing, but perform extra analyses on the resulting tree before extracting the entities. Their approach also goes further than detecting names of characters. The major difference however is the fact that their system does not store the data in RDF, which limits the interoperability of the system.

2. RPG on the Semantic Web

In the early days of the semantic web, there was some talk about the interesting combination of RPGs and the semantic web. Archived emails from the xml-dev[8] and the rdfweb-dev[9] mailing lists from 2003 and 2004 show that the idea to use RPGs on the se-

mantic web is about as old as the semantic web itself. After an extensive search very few papers were found that cover this subject. This seems to indicate that the interest in the subject has died out somewhat in recent years.

Chapter 3: Semantic Web Technologies

This chapter will give an overview of some of the technologies from the semantic web used during the design and implementation of the application that make this possible. Namely, *linked data*, *RDF*, *knowledge graphs*, *Turtle*, *HTML* with *RDFa*, *SPARQL* and *Solid* wil be covered.

1. Linked Data

On most of the web, the data provided by services and websites consists of just the requested resource, which makes it hard to find more data about the resource or discover related data. A proposal to mitigate this is *linked data*, which is data that contains links to other data so both the user and the computer can easily discover related data. A query to an API to get information about a person, might give back their birthday and full name, but also information about their significant other. In non-linked data, this field about the significant other could just contain the name of the significant other, which does not directly allow finding more information about this person. In linked data, this would contain a link to an information source about the person. This makes it possible to perform more complex queries that involve multiple data sources, without knowing where they are in advance.

2. RDF

In RDF, data is stored as *triples*. These triples consist of three parts, namely the *subject*, *predicate* and *object*. As the names imply, the subject is the resource being described, the object is the resource or data it is linked to and the predicate is the relation between the two. In other words, the subject and object are connected by the given predicate. For example, the triple `(Alice, knows, Bob)` means that Alice is linked to Bob with the "knows" predicate. In other words, this means that Alice knows Bob. Do note that this triple does not imply that Bob knows Alice, as that would be the triple `(Bob, knows, Alice)`. Literals can also be used as the object of a triple, such as strings or numbers. For example `(Alice, age, 23)` or `(Bob, nickname, "Bobby")`.

On the web, generic names such as `Alice` or `age` are not used, instead Uniform Resource Identifiers (URIs) are used to describe a resource or predicate. This is to make sure there is no ambiguity about what is being described. Everything gets its own unique URI. For example, Alice's identifier may be `http://mywebsite.com/Alice` and Bobs identifier may be `http://bobswebsite.com/card#me`. This way, every person or resource is uniquely identified.

Predicates are also referenced with URIs, so there is no ambiguity about which predicate is being used. To avoid having to reinvent every predicate or resource every time, there exist so-called *ontologies*, which are a set of predicates and resources designed for a specific purpose. These ontologies form the basis of the interoperability of the semantic web, because two services using the same predicate will know what the data represents, without having to interact with the other service. For example the `knows` predicate from before

could be the predicate from the “Friend of a Friend”-ontology (FOAF)[10] which has URI `http://xmlns.com/foaf/0.1/knows`. By using this URI, FOAFs definition of “knows” can be found:

> knows - A person known by this person (indicating some level of reciprocated interaction between the parties).

The FOAF ontology also provides a more thorough definition which provides more detail on when to use it and what the implications are.

Because URIs can become long and hard to read, a *prefix* is often used to shorten them. For example, the prefix `foaf` can be defined as being `http://xmlns.com/foaf/0.1/`, and instead of the whole URI of “knows”, the short version `foaf:knows` can be used, which is identical the full URI, but makes writing and reading triples easier. In this thesis, shorthand notation will mostly be used for predicates to increase legibility. The list of used ontologies can be found in List of Ontologies.

One ontology that is especially relevant to this thesis is the TTRpg ontology[11] that was created at the same time as this thesis was being written. It provides a set of URIs for describing tabletop RPGs and their characters, locations and actions. This ontology was created to provide a uniform way to describe this data. The ontology was made to be used in two ways, first to describe generic RPG-related data that was not bound to a specific game or type of RPG and second to create ontologies that describe a game such as Dungeons and Dragons[12] in detail, so the data can be made more accurate.

3. Knowledge Graphs

Triples can also be interpreted as being an edge between two nodes, where the edge has the label of the predicate and goes from the subject to the object. When combined into a graph, this is called a *knowledge graph*. As no knowledge graph can contain all the information in existence, it is often necessary to combine data from multiple graphs to get all the information needed. This is possible because each knowledge graph should use the same URI to describe a certain object.

On these knowledge graphs, so-called *reasoners* can be run, which are pieces of software that can make deductions about the data by looking at what is already defined and adding more information to the graph. For example, if `(person1, fatherOf, person2)` exists, it can be assumed that `(person2, childOf, person1)` should also exist. Reasoners can consist of simple rules as in the example, but can also exist of complex programs and/or use external sources for added information.

4. Turtle

Knowledge graphs can be serialized using a format called *Turtle*[13], a format that is both

machine and human readable. This section will not cover the whole specification, but provide context to be able to read snippets of data further in this thesis.

In its most basic form, Turtle files consist of three URIs separated by spaces which represent the subject, predicate and object respectively. URIs are written between triangle brackets `<>` and each line ends with a period. Defining prefixes is possible by writing `@prefix`, followed by the prefix, a colon and the URI it expands to. For an example, see Listing 1.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://mywebsite.com/Alice> foaf:knows <http://bobswebsite.com/card#me> .
```

Listing 1: Basic example of a Turtle file which defines the foaf prefix and a relation between Alice and Bob.

It is also possible to reuse the subject and the predicate in Turtle. To reuse the subject and predicate, end the triple with a comma instead of a period and type the new object after it. To reuse only the subject, end the triple with a semicolon and type the new predicate and object after it. These notations reduce the amount of duplicated URIs in the Turtle files and make them more legible. As an example, the two Turtle files in Listing 2 and Listing 3 represent the same information.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix schema: <http://schema.org/> .

<http://mywebsite.com/Alice> schema:name "Alice" .
<http://mywebsite.com/Alice> foaf:knows <http://bobswebsite.com/card#me> .
<http://mywebsite.com/Alice> foaf:knows <http://carol.organisation.org/about> .
```

Listing 2: Example Turtle file without shorthand notation.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix schema: <http://schema.org/> .

<http://mywebsite.com/Alice> schema:name "Alice" ;
                             foaf:knows <http://bobswebsite.com/card#me> ,
                                        <http://carol.organisation.org/about> .
```

Listing 3: Example Turtle file with shorthand notation that reuses the subject twice and the predicate once.

A final shorthand that is often used is the use of the string `a` instead of the predicate `rdf:type`. This way, we can type `<http://mywebsite.com/Alice> a foaf:Person` to define what type the resource

`<http://mywebsite.com/Alice>` has.

5. HTML + RDFa

To display web pages in a browser, most websites use HTML[14] which tells the browser what should be shown and how. Together with CSS this can make the data they show easily understandable for a human reader. For machine readers, this is significantly more difficult as the lay-out and structure of each document differs. This poses a problem for machine readers that analyze a site to simply get the data on it, but also for those that try make the site available for people with disabilities. For example, people that use a screen reader might run into problems if the screen reader cannot tell which parts it should read and which parts it should not read.

The latter problem can be helped with the use the appropriate HTML tags to indicate what semantic meaning each part of the document has. For example, marking the footer of a page as `<footer>` can tell the screen reader that that part may not be useful to the person if they are trying to read an article. It is also possible to add metadata to the tags, such as the `alt` tag for images, which tells the browser what is on the image. The Ghent University logo on the first page is tagged with the `alt` text “Logo Ghent University”, so screen readers can explain what is visible on the image.

In a similar way to the `alt` tag, RDF data can be embedded into HTML to explain what is meant by that piece of the page using RDFa[15]. For example, the title of this thesis is tagged with `property="foaf:name schema:name"`, which tells the browser that the title is “Creating an Extensible Semantic Web Framework for Annotating Role Playing Game Logs” according to the definition of both Schema.org and of Friend of a Friend. Programs can then easily extract this data by looking at the tags and provide extra functionality.

A commonly used example of this are previews that are shown when sharing a link on a chat application or on social media. If the website that is being linked provides the metadata in a way that the chat application or social medium understands, it can show the user a title, summary and/or a picture to indicate what the link will contain.

6. SPARQL

Knowledge graphs are only useful if it is possble to do something with them, which is why the query language SPARQL[16] was created. Like in Section 3.4, this section will not explain the entire specification, but explain enough to understand the SPARQL code in this thesis.

6.1. Querying

Querying a SPARQL database can be done with the SELECT keyword. The query itself consists of a format similar to Turtle, but with some URIs replaced with variables. These variables start with a question mark, directly followed by the name of the variable. The

SPARQL engine will then attempt to fill in these variables so that the resulting set of triples does exist in the knowledge graph. It could be that there are multiple options for variable bindings, in which case SELECT will return all possibilities.

A SELECT query consists of the word `SELECT`, followed by which variables that the SPARQL engine should return or a `*` if it should return all the variables in the query. After this comes a `WHERE` block, that consists of the word `WHERE` and the template between curly brackets. It is also possible to define prefixes before the SELECT query to make writing URIs easier. This is done by using the word `PREFIX`, the name directly followed by a colon and then the URI.

For example, the query in Listing 4 asks the SPARQL engine all the people that Alice knows.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?otherPerson WHERE {
    <http://mywebsite.com/Alice> foaf:knows ?otherPerson .
}
```

Listing 4: Example SPARQL SELECT query that selects every person that Alice knows, using variable ?otherPerson.

If the SPARQL engine supports multiple different graphs, it is possible to add one or more `GRAPH` blocks inside the `WHERE` block to define which graph the data should be queried from. Variables can be used between different `GRAPH` blocks. For example, if an application has a public and a private graph where the private graph contains sensitive information such as e-mail addresses and the public graph contains information such as friends and name, the e-mail addresses of all the people Alice knows can be queried with the query in Listing 5 (assuming access to the private graph).

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?email WHERE {
    GRAPH <http://myapp.com/public> {
        <http://mywebsite.com/Alice> foaf:knows ?otherPerson .
    }

    GRAPH <http://myapp.com/private> {
        ?otherPerson foaf:mbox ?email .
    }
}
```

Listing 5: SPARQL SELECT query with multiple graphs where the email address is queried from a private graph for each person that Alice knows according to the public graph.

6.2. Adding data

The keyword `INSERT` is used to add data to the knowledge graph. The easiest way to insert data is using `INSERT DATA` followed by one block containing the data to be inserted.

See Listing 6 for an example.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

INSERT DATA {
    <http://mywebsite.com/Alice> foaf:name "Alice".
}
```

Listing 6: Example INSERT query that adds the data that Alice has the name “Alice”.

However, if the data inserted depends on the data already in the database, the query can also consist of an `INSERT` and a `WHERE` block. The SPARQL engine will first bind the variables in the `WHERE` block and fill these in in the `INSERT` block and add those to the database. For example, the query to add the fact that Bob knows everyone with the nickname “Jimmy”, can be seen in Listing 7.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

INSERT {
    <http://bobswebsite.com/card#me> foaf:knows ?jimmy .
} WHERE {
    ?jimmy foaf:nick "Jimmy" .
}
```

Listing 7: Example INSERT WHERE query that inserts the fact that Bob knows everyone who has a nickname of “Jimmy”.

7. Solid

Solid is a system that provides the ability for users to create their own *datapods*, or *pods* for short. It was started by the inventor of the web, Sir Tim Berners-Lee as a way to counteract the current trend of applications to keep a users data in their own data vaults. This way, users would have their data in a storage medium they control and applications read from those pods when they need the data. This cuts the current hard-wiring between application and data and replaces it with an environment where data and applications are handled separately.

Chapter 4: NLP

Humans write text in a way that makes sense to humans using things like sentences, words and punctuation. This allows the expression of complex topics and concepts, in many different ways. Humans can extract the meaning of these sentences fairly easily, but for a computer this task is significantly harder. The study of analyzing this natural language using computers is called Natural Language Processing (NLP) and is utilized in many different fields such as commerce, where an automated response may be sent to the client if they ask for a common support question. This section will explain two major NLP techniques, namely *constituency parsing* and *named entity recognition*.

1. Constituency Parsing

The first step to understanding a sentence is to figure out what function each word has in the sentence. To do this, the sentence gets parsed into a tree where each node represents part of the sentence and what function it performs. Nodes can have child-nodes that describe that part of the sentence more specifically, usually by splitting it up into smaller parts. Some example labels are:

- `S`: Sentence
- `N`: Noun
- `V`: Verb
- `NP`: Noun Phrase (group of words that serves the function of a noun)
- `VP`: Verb Phrase (group of words that serves the function of a verb)

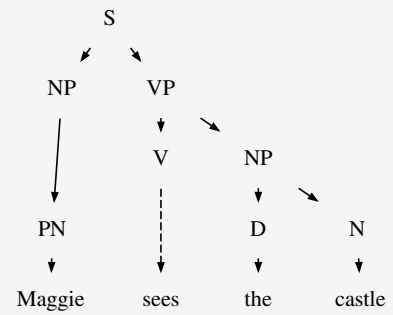

Figure 9: Constituency tree of the sentence “Maggie sees the castle”. Each node in the tree represents a part of the sentence and its children represent a further subdivision. The label on the node indicates which function that part of the text serves.

Once this tree of the sentence is constructed, reasoning over it in an abstract manner becomes possible. For example, if changing the value of a `NP`-node in a sentence with another noun phrase will make sure the sentence remains grammatically valid. These operations are significantly harder on an sentence that has not been parsed, as it is not known in advance what function each word performs. An example parse tree can be found in Figure 9.

2. Named Entity Recognition

Named entities are all entities that can be directly referenced by a name. This includes proper nouns such as Hildegarde or Jeffrey, but not general terms such as “the throne room”. The process of extracting these named entities from a sentence is called *named entity recognition* (NER). For example, the sentence

> Hildegarde wants to hide Jeffrey in the throne room.

can be tagged as

> $[\text{Hildegarde}]_{\text{person}}$ wants to hide $[\text{Jeffrey}]_{\text{person}}$ in the throne room.

There are many ways to perform NER, but this thesis makes use of constituency trees. The constituency tree of the example sentence can be seen in Figure 10.

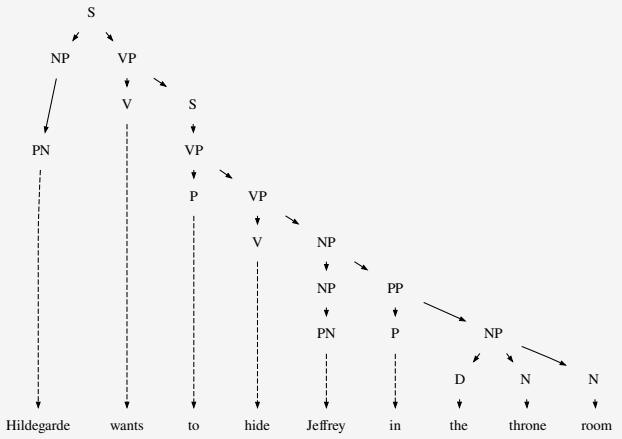

Figure 10: The constituency tree of the sentence “Hildegarde wants to hide Jeffrey in the throne room”.

From this constituency tree, all the proper nouns (PN) and noun phrases (NP) can be found, as well as their value. These values can be matched with all the characters that are known and check if any of their names match the value. If this is the case, the tag can be added. Analyzing the constituency tree in Figure 10 in this way and extracting the sentence from it, this becomes.

> $[\text{Hildegarde}]_{\text{http://calippo.com/hildegarde}}$ wants to hide $[\text{Jeffrey}]_{\text{http://calippo.com/jeffrey}}$ in the throne room.

Once these tags are known, they could be used to include clickable links to the characters or allow the front-end to filter only sentences that contain a given character. This is especially useful if a character has multiple different names, such as a princess named Helena that goes undercover as Diede, or a character named “Brother Jacoba” that is sometimes called “Jacoba” and other times “The Brother”.

Performing the named entity recognition this way also avoids tagging words that are not names. For example, a character with a name that could also be a verb such as “Flip” will not be tagged in sentences like “I flip a coin”, as the word “flip” in that sentence is a verb and not a proper noun.

Chapter 5: Architecture

Overview

This section will lay out the design of the application and what choices were made to get to the final result. These choices are often the question of how to save the data in a way that it can be easily reused and what ontologies to use. The overview in Figure 11 shows all the steps schematically, which are the following:

1. Extract pieces of text from raw files.
2. Split pieces of text into sentences.
3. Parse sentences into constituency trees.
4. Add metadata to the constituency trees using a Solid pod.
5. Reconstruct the original sentences, enriched with metadata.

Each of the following sections will go into more detail on one of these steps.

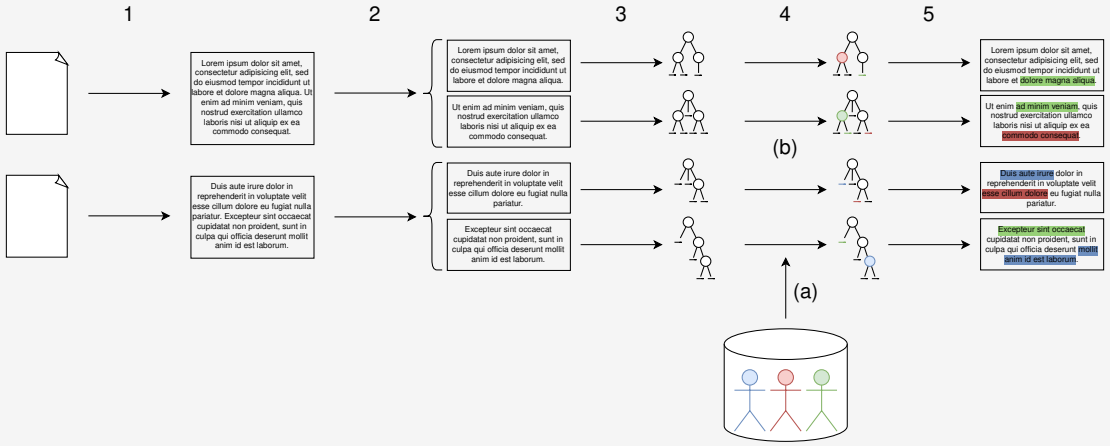

Figure 11: Overview of the design of the application. First, the raw files get transformed into pieces of text (1). Then, the pieces of text get split into sentences (2) which get parsed into constituency trees (3) and tagged with which characters they reference (4b) with data from a Solid pod (4a). Finally, the original texts are reconstructed in a semantically meaningful way (5).

1. Extracting Text From Files

The first step in the process is to acquire the text that is contained in the player logs in structured data. As these logs can come in many different formats, it is important to first turn them into a known structure, independently of their original format. There are many ways to store plain text in RDF, but the one used in this thesis uses the definition by IOLite, namely `iol:Text`.

IOLite[17] (prefix `iol`), short for "Information Objects ontology lite" is an extension of the DOLCE Ultralite[18] (prefix `dul`) upper-ontology. DOLCE Ultralite is an upper-ontology that provides concepts that can be reused by other ontologies so there is interoperability among these ontologies. DOLCE Ultralite focuses on physical and social concepts and IOLite is an extension that, as the name implies, allows the representation of information objects. An *information object* is, according to the definition by DOLCE Ultralite:

> A piece of information, such as a musical composition, a text, a word, a picture, independently from how it is concretely realized.

Which corresponds to what is needed to store the texts. To indicate which value the piece of text has, the predicate `rdf:value` is used. An example is shown in Listing 8.

```
<http://pantheonparty.com/text/1> a          iol:Text;
                                  rdf:value "Beowulf sees Johan on top of the
                                  hill. He draws his sword and walks closer."@en.
```

Listing 8: Turtle representation of the text "Beowulf sees Johan on top of the hill. He draws his sword and walks closer."

Since each type of data requires a specific approach, this was done manually for each set of logs. Most of the time, an `iol:Text` was created for each paragraph, but some sources did not have paragraph, so other divisions were made too.

2. Splitting Text Into Sentences

The second step to analyzing text was to split the text up into sentences that can be analyzed separately. While this step in itself is pretty straight-forward, it is important to split the possibly large pieces of text up into smaller, manageable pieces. The main design consideration was how to model the data in linked data.

As the previous step uses `iol:Text` to store raw, untreated text, this step uses that format to read in text as well. It uses `iol:Sentence` for sentences extracted from this text. The value of the `iol:Sentence`s is indicated with the predicate `rdf:value` and to link the `iol:Text` and `iol:Sentence`s, the predicate `dul:hasComponent` is used. The piece of text in Listing 8 from the previous part will turn into the data shown in Listing 9.

```
<http://pantheonparty.com/sentence/1> a iol:Sentence;
            rdf:value "Beowulf sees Johan on top of the hill.".

<http://pantheonparty.com/sentence/2> a iol:Sentence;
            rdf:value "He draws his sword and walks closer.".

<http://pantheonparty.com/text/1> dul:hasComponent <http://example.com/sent/1>,
                                                   <http://example.com/sent/2>.
```

Listing 9: Turtle representation of two sentences that originated from one text and how their attributes are stored.

3. Constituency Tree Parsing

The next step in analyzing sentences is parsing the sentences into a constituency tree. Once this constituency tree is constructed, it needs to be converted into RDF, for which a data model needs to be constructed.

The main ontologies for this task are the NLP Interchange Format 2.0 core ontology[19] (prefix `nif`) and the Ontologies of Linguistic Annotation (OLiA) [20] (prefix `olia`). Both of these are commonly used when representing NLP data in RDF.

The NLP Interchange Format (NIF) is a format that provides interoperability between different NLP tools, by providing specifications, ontologies and software. For this step, mainly the NIF core ontology will be used, as this provides most of the predicates needed. More specifically the following predicates will be used:

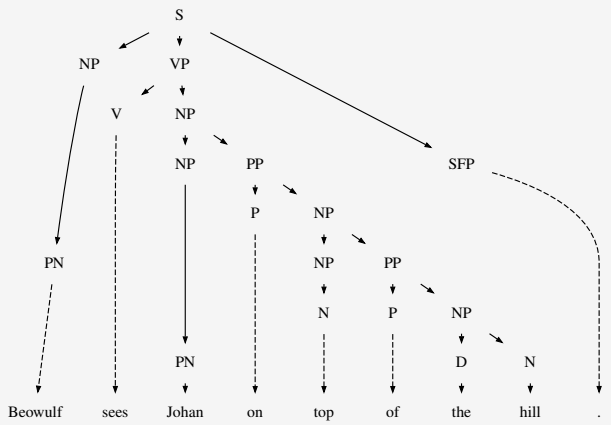

Figure 12: The constituency tree of the sentence “Beowulf sees Johan on top of the hill.”.

- `nif:isString`: Indicates the value of a node (and its children).
- `nif:posTag`: The Part of Speech tag, as an OLiA tag (see below).
- `nif:subString` and `nif:superString`: Indicates that the object is a part of the subject and vice versa.
- `nif:beginIndex` and `nif:endIndex`: Indicates where a substring starts and ends in the superstring, both zero-based, inclusive indices.

The Ontologies of Linguistic Annotation (OLiA) is a set of definitions for many different types of NLP annotations. In this case, their definitions of the part of speech (POS) tagging of words are used to tag the constituency trees.

One of the sentences from the previous examples “Beowulf sees Johan on top of the hill.”, leads to the constituency tree in Figure 12. Since printing the whole tree in Turtle format would be too long, only two nodes are serialized. The first one is the root node, containing the entire sentence. The second one is the noun phrase “the hill”. Both can be seen in Listing 10. As can be seen, the major difference is the fact that non-root nodes have the predicates `nif:beginIndex`, `nif:endIndex` and `nif:superString`. The latter node has a begin and end index of 3 and 10 respectively, as the parent node represents

the value “of the hill”, of which we need to take characters 3 to 10 (inclusive) to get “the hill”.

```
# Root node
<http://pantheonparty.be/constituency-node/1> a nif:String, iol:Sentence;
            nif:isString "Beowulf sees Johan on top of the hill.";
            nif:posTag olia:Sentence;
            nif:subString <http://pantheonparty.be/constituency-node/2>,
                          <http://pantheonparty.be/constituency-node/3>,
                          <http://pantheonparty.be/constituency-node/4>.

# Noun Phrase node

<http://pantheonparty.be/constituency-node/10> a nif:String;
            nif:isString "the hill";
            nif:beginIndex 3;
            nif:endIndex 10;
            nif:posTag olia:NounPhrase;
            nif:superString <http://pantheonparty.be/constituency-node/9>;
            nif:subString <http://pantheonparty.be/constituency-node/11>,
                          <http://pantheonparty.be/constituency-node/12>.
```

Listing 10: Turtle representation of the root node of a constituency tree and a child node in the RDF representation.

4. Recognizing Entities using Solid Data

This step consists of two separate tasks, namely getting the Solid data and extracting the entities. Both will be described below.

4.1. Getting Solid Data

The user should be able to store the data of their characters and campaigns in their personal Solid pod. This way, they have control over their own data and can link to other pods to create a decentralized storage of the characters. Before the data can be analyzed, a local copy of this data is needed that can be accessed by the SPARQL engine for easier processing.

The Community Solid Server (CSS) has a functionality that allows the use of client credentials, which removes the need to store the users password to log in, but utilizes a generated key that can be revoked by the user when needed. With these credentials, the application can request a token id and secret that allows it to make request authenticated as the user for a certain period of time.

By starting from a user-provided root and recursively fetching all resources in the pod, all data can be stored to a quad store. A quad store is useful because this data model orders triples using named graphs. By using these named graphs, we can keep track of the source of each triple. By constructing a SPARQL INSERT query, all the data can be added to the central database.

4.2. Recognizing Entities

With all the previous steps set up, a basic form of Named Entity Recognition can be performed. In this step, each proper noun or noun phrase in the constituency tree is matched to the names of the characters that are currently known in the database.

Names can consist of one word, such as “Heksina” or “Serafina”, which are just one proper noun. Names can also consist of two words, such as “Lady Ghost”, in which case the full name is a noun phrase and needs to be matched as a whole. Because of this, both proper nouns and noun phrases need to be checked to get all the references.

To link the node in the constituency tree and the character it references, the `itsrdf`[21] ontology is used, which was made by W3C to allow mapping of the Internationalization Tag Set[22] into RDF data. This step only uses the `taIdentRef` predicate, which indicates which **ident**ifier the subject **ref**erences to.

In the example sentence “Beowulf sees Johan on top of the hill.”, the nodes representing “Beowulf” and “Johan” will get a `itsrdf:taIdentRef` link to their respective character. We can see this for the node containing the proper noun “Beowulf” in Listing 11.

```
# other NIF tags have been removed for brevity
<http://pantheonparty.be/constituency-node/20> a nif:String;
                    nif:isString "Beowulf";
                    itsrdf:taIdentRef <http://pantheonparty.com/beowulf>
```

Listing 11: Representation of the node containing the proper noun “Beowulf” with a reference to the character Beowulf.

5. Reconstructing Sentences With Metadata

The final step is to produce a result by recombining the different constituency nodes and their links to characters into legible HTML code. This process starts by finding the root node of a sentence and iteratively recombining the child nodes. To get the original sentence, it suffices to concatenate the values of all the leaf nodes into one sentence. The original location of whitespaces can be determined using the begin and end indices of the nodes. The nodes with a link to a character gets a separate `<span>` that contains RDFa to indicate that it references a `ttrpg:Character` and a link to which character. We can also add a `title` attribute that allows the user to hover over the word an get more information. As an example, the sentence from the previous step would become the following HTML:

```html
<p>
    <span typeof="ttrpg:Character"
        resource="http://pantheonparty.com/beowulf"
        title="Beowulf (Pantheon Party)">
        Beowulf
    </span>
    sees
    <span typeof="ttrpg:Character"
        resource="http://pantheonparty.com/johan"
        title="Johan (Pantheon Party)">
        Johan
    </span>
    on top of the hill.
</p>
```

Listing 12: HTML representation of the tagged sentence “Beowulf sees Johan on top of the hill.” with a title attribute for both characters referenced and a resource tag to link it to the URI of the character.

When rendered this becomes:

Beowulf sees Johan on top of the hill.

Note: In the HTML version, users can hover over the names of the character to get a hint box showing the name of the character and the campaign it originates from. For example, when hovering over Beowulf, the text “Beowulf (Pantheon Party)” shows up. In PDF, this is not available, but a screenshot is shown in Figure 13.

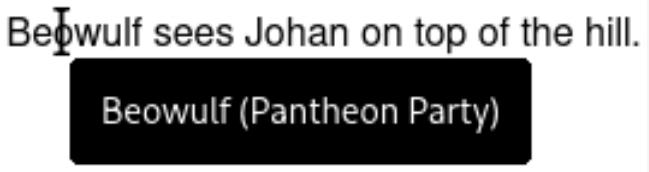

Figure 13: Preview of the hint box in HTML that would be shown when hovering over the name of a character.

Chapter 6: Implementation

Overview

This section will explain the technical aspect of the application to implement the steps designed in the previous chapter. The first section will show how the data was converted into pieces of `iol:Text`. The second section will explain the base of the application, namely *semantic.works*. Section 3 to section 6 will explain four different services that implement a reasoner that perform a part of the analysis, namely *sentence-service*, *constituency-tree-service*, *solid-sync-service* and *named-entity-recognition-service*.

1. Data

To test the application, a set of RPG game logs was needed to use as input for the application and check the results. Using existing game logs allows testing how the system would function on real-world data. This set of logs comes from myself, from friends and even some from an RPG IRC channel.

All personally identifiable data of the players playing the character was removed before processing. For the campaigns, permission was asked to use it for data to test the applicaion, but not to republish the data. The examples used in this thesis come from campaigns where explicit permission was asked of all the players to use their character and storyline.

To process the data, different Jupyter[23] notebooks were used for each campaign to easily process the data in steps and perform checks during the process. As this contains links to private data, these scripts are not public, but the general idea of how they were processed will be explained.

In the end, 9 different campaigns were analyzed, with a total of 62 documents containing logs. This resulted in 4138 pieces of text that could be processed by the application.

While processing the data, different challenges came up to get the data from their raw source to the correct RDF format. The rest of this section will highlight three different challenges that came up. Each subsection will cover one of them, namely the format of the logs, the language they are written in and their phrasing.

1.1. Format

Almost every GM used a different format to keep logs which made it difficult to find a way to extract pieces of text from each of the different formats. The rest of this subsection will describe the formats and how they were processed.

1.1.1. Markdown

Markdown is a plain-text format that was made to create documents that can easily be turned into HTML documents and are still legible in their source format. In its most basic form, Markdown documents contain normal text where each paragraph will be shown as is. It allows adding markup such as *italic* or **bold** by surrounding the word with respectively one or two asterisks, so `*this*` becomes *this*. Titles can be inserted, up to 6 levels deep by adding one or more `#` before the title. The more are added, the lower the title is. For example the title of this subsection is `### Format`, as it is 3 levels deep (chapter title, section title and subsection title). Markdown also allows links to be specified by using `[text](url)`, where a link will be shown with “text” as display text and a link to “url”.

To get text from raw Markdown, it suffices to take the text as is and remove all special markup characters such as `*`. Links can be removed using a regular expression. This cannot be done by removing all brackets, as they may be used in a sentence. Once these are removed, each paragraph, separated by two newlines, can be processed as a piece of text.

The Markdown documents used to analyze came from many different sources, which introduced slight differences in how they were processed. Some came as raw Markdown files which just needed to be opened and read. Others were stored on Hackmd.io[24], which required an HTTP request to get their data, but were otherwise identical to raw files. One campaign used a GitHub wiki to keep track of their campaign, so the repository had to be cloned in a folder to get the raw Markdown files. Finally, one campaign used Obsidian[25], whose format is mostly identical to Markdown, but has some extended syntax that needs to be removed as well.

1.1.2. MediaWiki

MediaWiki[26] is a system that was originally developed for Wikipedia, but is now used in many different places. One of the GMs kept their logs on a self-hosted instance of MediaWiki which also stored information on characters and other related information. This allows the GM to create links to information pages about a specific subject, so they can more easily find extra information if needed.

The format of MediaWiki is, just like Markdown, at its basis a plain-text format. It allows adding links to pages on the same wiki by using `[[Subject]]`, which would link to the page named `Subject` on this wiki. It also allow markup, *italic* and **bold** are done by surrounding the text with respectively two or three backticks. For example: ```` ```text``` ```` becomes **text**.

To make this into plain text, a similar approach to markdown was followed by removing special characters using regular expressions.

1.1.3. Google Documents

Some campaigns kept their logs in a Google document. To query this, an existing Python library was used that provided helpers for a users login and access to a Google document. This needed a Google application that had OAuth 2.0 set up and kept the tokens for this in a secret file.

The login flow asks the user to log in with a Google account. In this case this should be an account that has access to the document to be analyzed. With a logged in client, the document can be requested. The library returns the document as a Python object which can be queried like any other dictionary. To get all the necessary paragraphs, the application needs to iterate over everything in the content of the body and check if that piece of content contains a paragraph. If it is, it needs to check the parts in that paragraph and combine all text from it.

By using the “textRun” element, the text is retrieved without any markup which removes the need to strip anything from it to get the final texts.

1.1.4. Webblog

One campaign kept a blog online with their campaign logs. This made processing a bit more difficult, as the HTML tags were not always consistent. Using the BeautifulSoup[27] library, the HTML could be parsed and all paragraphs from the page could be extracted. This mostly worked, but it was still required to filter out titles and other interjections that

were not part of the storyline.

1.2. Language

Some of the campaign logs were written in Dutch as that is the native language of some of the GMs. While it is theoretically possible to use models that support Dutch, this research focuses on English only, as the thesis itself would be written in English. Therefor, the texts needed to be translated into English before they could be stored.

To translate the pieces of text to English, a Python library was used that interfaced with Google Translate. While still imperfect, the translations were good enough to use for further processing. The library was however fairly inconsistent and would often just return the original, Dutch text instead of translating it. It would give no error message, so a manual check was added to see if the texts were translated or not. After some time it turned out that it was due to rate limiting on the Google Translate API. It would only allow a certain amount of requests per hour, so if the application ran into this issue, it waited some time before retrying. This made the translation step take quite some time.

Besides this, the translation library would often translate a sentence incorrectly. It resulted in a significant loss of quality. For example, names would be translated in strange ways, such as the name “Heksina”, that would often be translated into “Weksina”, making it impossible to recognize the original character without manual intervention.

To keep the original text, as well as the translated version, language annotations were used to indicate the language of the string. For English sentences, only the English sentence was saved using the `en` language tag, but for Dutch sentences, the original sentence was stored as well with the `nl` language tag.

1.3. Phrasing

Keeping logs during a session is an intensive task, especially when combined with playing a character or managing a session as GM. Because of this, GMs and players often choose to not keep logs in as full sentences, but only write some keywords or drop words from the sentence. This makes it easier to write and people can understand it well enough in context, but analyzing this becomes a real challenge. Especially constituency tree parsing becomes difficult when the sentence is not a valid sentence. This can easily result in incorrect parsing and thus produce incorrect results.

2. Semantic.works

The base of the application is the Semantic.works framework[28], which is a framework that revolves around a system of microservices that interact with a central triple store. Each of the microservices implements a reasoner that processes the data in the knowledge graph. This way, services can be reused easier, as they only communicate through the central database. Semantic.works was formerly named “mu.semte.ch”, so many of the services it uses are still using the `mu-` prefix in their name.

2.1. Semantic.works middleware

The core of Semantic.works are three services: the identifier, the dispatcher and the database. These allow services to communicate and provide a way to process requests from the frontend. An overview can be seen in Figure 14.

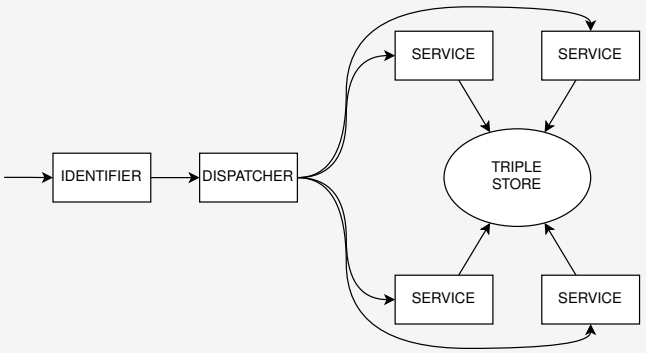

Figure 14: Overview of the basic Semantic.works framework. The identifier takes in requests from the frontend and passes them on to the dispatcher, who in turn forwards it to the right service. Services mainly communicate through the central triple store.

2.1.1. Identifier

The entry point to the backend is the *identifier*, which takes in the request from the frontend and finds out to which session it belongs by sending and reading a cookie which contains a unique URI for each session. In most cases, this session corresponds to a browser-tab from which the user made the request. It attaches the session id to the request using an HTTP header before sending it to the dispatcher.

2.1.2. Dispatcher

The *dispatcher* takes a request from the identifier and determines which microservice it should be forwarded to for processing. This way, different microservices can define an API endpoint with the same name inside their container, without causing problems. It does so by matching the request URL to a preset list of paths and their respective microservice.

2.1.3. Database

The *database* consists of an OpenLink Virtuoso triplestore, which allows services to read and write data using a SPARQL endpoint. It also supports the usage of different graphs for storing triples. The database is available to every service and serves as the primary way of

communication.

2.2. Provided services

Some services are used in many different applications, as they often have similar requirements. Because the Semantic.works stack revolves around reusing services, many common services are available. This application uses three of these, namely *mu-authorization*, *delta-notifier* and *mu-cl-resources*.

2.2.1. Mu-authorization

The *mu-authorization* service sits between the services and the database and adds authorization to the requests. It checks if the user belonging to the session provided by the identifier has the right to read or write the data requested in the SPARQL query. To allow different users to be able to read and write data, the service makes use of different graphs. When a write occurs, it writes this data to all graphs that correspond to a group that has read access to this data, so that if a read occurs, it can simply read from the right graph to return the data that group is allowed to read. A configuration file defines which users can read and/or write to which graphs.

As this service also provides a SPARQL endpoint at the same endpoint as the database, services can simply be configured to send their requests to the mu-authorization service instead of to the database directly and, given that they have the correct permissions, can simply send requests as if they were talking to the database directly.

Mu-authorization also provides a way to have services get notified when a write-request is made, by sending the changes or *delta*s to a service that wants to receive those changes.

2.2.2. Delta-notifier

The *delta-notifier* is a service that takes the deltas from the mu-authorization service and forwards them to the interested services. The delta-notifier also allows each service to specify which deltas they are interested in, so they don't have to receive all the deltas all the time. This allows the services to know when something has changed without having to query the database continuously.

The deltas are sent in JSON format as a list of objects. Each of these objects represents a change in the data and has two fields, `inserts` and `deletes`, which both consist of a list of objects representing triples that are inserted or deleted respectively. These triples have three fields `subject`, `predicate` and `object` with as value another object that follows the specification for how to encode RDF terms from SPARQL[29]. An example is found in Listing 13

```
[
  {
    "inserts": [
      {
        "subject": {
          "type": "uri",
          "value": "http://pantheonparty.com/sahara"
        },
        "predicate": {
          "type": "uri",
          "value": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
        },
        "object": {
          "type": "uri",
          "value": "http://w3id.org/TTRpg#PlayableCharacter"
        }
      }
    ],
    "deletes": [
      {
        "subject": {
          "type": "uri",
          "value": "http://pantheonparty.com/sahara"
        },
        "predicate": {
          "type": "uri",
          "value": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
        },
        "object": {
          "type": "uri",
          "value": "https://dbpedia.org/ontology/desert"
        }
      }
    ]
  }
]
```

Listing 13: Example of a delta where the rdf:type of a resource changes.

2.2.3. Mu-cl-resources

To make it easier for existing frontend tools to interact with the Semantic.works framework, *mu-cl-resources* was created. It provides a way to specify a mapping between the data in the triple store and a JSON:API[30] endpoint. The developer can specify what the different resources in the API look like and how these correspond to triples in the database. Mu-cl-resources will then provide an API that allows all operations to read/write these as objects and automatically map this to changes in the triple store.

Mappings can be defined using a Lisp and/or JSON configuration. While they provide the same functionality, this section will be focusing on the JSON version. This version requires the following fields for each type:

- `name`: The name of the resource.
- `class`: A URI describing the type of this resource.

- `attributes`: The attributes this object has, each consisting of
  - `type`: The datatype of the attribute, such as `string` or `number`.
  - `predicate`: The URI of the predicate that corresponds to this field.

- `relationships`: Which links this type has to other objects.
  - `target`: The name of the type the relationship links to.
  - `predicate`: The URI of the predicate that corresponds to this relationship.
  - `cardinality`: The cardinality of the relationship, either `one`, when there is only one such relationship for each object or `many`, when there are multiple.
  - `inverse`: Indicates if this relation corresponds to the inverse of `predicate`.

For example, a mapping for authors and books can be seen in Listing 14 which will map the data in Listing 15 to Listing 16

```
{
  "version": "0.1",
  "prefixes": {
    "schema": "http://schema.org/"
  },
  "resources": {
    "book": {
      "name": "book",
      "class": "schema:Book",
      "attributes": {
        "title": { "type": "string", "predicate": "schema:headline" }
      },
      "relationships": {
        "author": {
          "target": "author",
          "predicate": "schema:author",
          "cardinality": "one",
        }
      }
    },
    "author": {
      "name": "author",
      "class": "schema:Person",
      "attributes": {
        "name": { "type": "string", "predicate": "schema:name" }
      }
    }
  }
}
```

Listing 14: Example mu-cl-resources configuration describing authors and books.

```
<http://example.com/mybook> rdf:type schema:Book;
                            schema:headline "My Diary";
                            schema:author <http://example.com/robbevanherck>.

<http://example.com/robbevanherck> rdf:type schema:Person;
                                   schema:name "Robbe Van Herck".
```

Listing 15: Example RDF data for books with one book defined with title “My Diary” and one author with name “Robbe Van Herck”.

```
{
  "data": {
    "attributes": {
      "title": "My Diary",
    },
    "id": "4263c1e6-beb5-417d-8428-24cb2a2d96f0",
    "type": "book",
    "relationships": {
      "author": {
        "links": {
          "self": "http://example.com/robbevanherck",
        }
      },
    }
  },
  "links": {
    "self": "http://example.com/mybook"
  }
}
```

Listing 16: The RDF-data from above mapped to JSON:API using the configuration described before.

3. Sentence service

The first service that was implemented for the application was the *sentence-service*, which has the simple task of waiting for new pieces of text to be added to the database and split them up into separate sentences. The task itself is pretty straight-forward, but it serves as a reminder that reasoners don't need to be complex to be useful. Also, since it was implemented first, it laid the foundation for how other services would interact with the database, receive deltas, etc.

The service makes use of Semantic.works' Python template [31], which provides some helpers like a function to easily perform SPARQL queries to the database and a pre-configured Flask app. This way, only a file which contains the implementation of the service needs to be provided and the template will provide the helpers, start up the application, etc.

```
function handle_delta(delta) {
    text_uris = []
    values = {}
    for each inserted triple in delta {
        if (triple.predicate == "rdf:type" &&
            triple.object == "iol:Text") {
            text_uris.add(triple.subject)
        }
        if (triple.predicate == "rdf:value") {
            values[triple.subject] = triple.object
        }
    }

    for each uri in text_uris with a value in values {
        for each sentence in nltk.sent_tokenize(value) {
            store sentence
        }
    }
}
```

Listing 17: Pseudo code overview of the sentence-service.

The service consists of one endpoint, namely `/.mu/delta`, which is the endpoint that most Semantic.works services use for receiving deltas from the delta-notifier. When this endpoint gets a POST request, the service parses the deltas and creates a list of sentences it should analyze. It keeps a list of all the URIs that it knows are of type `iol:Text` and also keeps a dictionary that maps URIs to their `rdf:value`, if they have any. We have to keep both separately to later match the URIs that have a `rdf:value` and are known to have type `iol:Text`, as these are the texts we will be processing.

We find these by looping over all the inserted triples and checking the predicate. If the predicate is `rdf:type`, it checks if the object (and thus the type of the subject) is `iol:Text` and if so, it adds it to a list of `iol:Text`s. If the predicate is `rdf:value`, it creates an entry in the dictionary with the URI of the subject as key and the string in the ob-

ject as value.

The actual separation of sentences is done with the Python library NLTK [32]. More specifically, the `sent_tokenize` function is used, which splits up the text into sentences. Each of these sentences will then be saved to the database in the correct format. To make it accessible by mu-cl-resources, a `mu:uuid` needs to be added, which provides a unique identifier.

4. Constituency Tree Service

The second service in the application is the *constituency-tree-service*, which takes sentences in the database and performs constituency tree parsing on them.

Like the sentence-service, constituency-tree-service is also based on the mu-python-template and contains one endpoint `/.mu/delta` to receive delta notifications.

The sentences are analyzed using CoreNLP[33], which is a Java program that can do many NLP tasks, including parts of speech tagging and constituency tree parsing. Because the constituency-tree-service is written in Python and CoreNLP is written in Java, there is no way to directly interface with it. However, NLTK provides a helper class which allows using the CoreNLP tools from within Python using its API.

To make sure the CoreNLP server works on every system, a Docker container running an instance of CoreNLP is added to the system that does not interact with the database or any of the middleware services directly, but services can choose to talk to the server if they need CoreNLP analysis.

```
function handle_delta(delta) {
    database_graph = connect_to_database()
    sentence_uris = []
    values = {}
    for each inserted triple in delta {
        if (triple.predicate == "rdf:type" &&
            triple.object == "dul:Sentence") {
            sentence_uris.add(triple.subject)
        }
        if (triple.predicate == "rdf:value") {
            values[triple.subject] = triple.object
        }
    }

    for each uri in sentence_uris with a value in values {
        tree = corenlp.parse(value)
        store_recursive(database_graph, tree)
    }
}

function store_recursive(database_graph, tree, start_index=0) {
    database_graph.add_triple(tree, "rdf:type", "nif:String")
    if !database_graph.has_uuid(tree) {
        database_graph.add_triple(tree, "mu:uuid", generate_uuid())
    }

    current_character = start_index
    if tree.has_children {
        for child in tree.children {
            while (current_character.is_whitespace()) {
                current_character += 1
            }
            characters_processed = store_recursive(database_graph, tree, current_
            current_character += characters_processed
        }
        value = substring of sentence from start_index to current_character
    } else {
        value = tree.value
    }

    pos_tag_uri = lookup_pos_tag_uri(tree.pos_tag)

    store current node with start, end, pos_tag_uri and value
}
```

Listing 18: Pseudo code overview of the constituency-tree-service.

Once the application starts up, it makes a connection to the CoreNLP server and waits for delta notifications. Similar to sentence-service, it finds all URIs that have a `rdf:type` of `dul:Sentence` and have a `rdf:value`. All of these get fed through the parser using NLTKs `CoreNLPParser.parse` function. This returns the parsing as a dict.

The communication with the database goes through `rdflib`, which provides a way to use the same functions as for in-memory graphs, but with the data being stored automati-

cally to a SPARQL triplestore. The connection to the store is opened right after analyzing the sentence and passed on to each function call so the calculated data can be saved immediately.

The way the constituency tree is processed uses a recursive approach that starts by processing the root node and process the rest of the tree down from there.

The first thing that needs to be done is adding the type of `nif:String` to the node, if the node already exists it suffices to add a triple with predicate `rdf:type` and object `nif:String`, otherwise a uuid needs to be generated and stored so mu-cl-resources recognizes it. The `rdf:type` triple gets stored first as this way, mu-authorization knows its type and knows that the permissions for it allow the application to write these triples. Otherwise, all triples before the type would be rejected a it doesn't know that it is a `nif:String`, which can be writen to without special authorization.

After that, a check is performed to see if the node has children, if this is the case, the child is processed in the same way. Keeping a running count of how many characters are processed allows determining what the start and end indices are for each child node. It also makes it possible to find out which part of the sentence this node covers. Before processing a child node, white space is skipped, otherwise spaces would be considered part of a word. Using a recursive call, each child node gets processed which returns the number of characters processed and the URI where the child was stored. With this, `nif:beginIndex` is known to be of characters processed so far and the `nif:endIndex` is the `nif:beginIndex` plus the number of characters covered by the child node, minus 1 to be an inclusive index. The `nif:superString` for each child node is the current node and the child node is a `nif:subString` of the current node.

Finally, the POS tag is stored using an OLiA tag, using a mapping from CoreNLP labels to OLiA URIs. CoreNLP uses Penn Treebank labels, which OLiA supports. For this, a Python dict with these mappings was set up which contains the 58 entries that occur in the dataset, which should cover the majority of English sentences.

Penn Treebank labels can also contain *function tags*, for example, a verb might be vocative, which would result in the tag `VB-VOC`, where `VB` indicates it is a verb and `-VOC` that it is vocative. In this application, this function tag is not stored as it not necessary further down the line, so it is stripped from each tag by splitting on the character "-" and taking the first group. If future applications do want to use this tag, it could be added by checking the other groups after splitting and changing which tag to give depending on all values instead of just the first.

5. Solid Sync Service

To make processing data from the users Solid pod more easy, *solid-sync-service* was created. This service allows data to be mirrored from a Solid pod in the database, so it can be queried like data that is already in the database. Right now, it supports mirroring one pod, but it is possible to expand this to include multiple pods and things like per-pod authorization.

The service currently only works with Community Solid Server (CSS) as this version allows the use of client credentials. These allow a user of a Solid pod to create a set of tokens that can authenticate the user without having to store their username and password in the app. This is especially useful for backend applications that don't have the option to show the user the login prompt in a browser for authentication. These tokens are implemented as a temporary measure to allow clients to connect without needing to know the users plain-text password and might be changed in the future.

The user can request a new token by sending a POST request to the credentials path. This path can be found by interacting with the identity provider (IDP) and looking at the URIs in the response. The user should provide their login details once, as well as the name of the token. As the user can do this manually, the service that uses the token doesn't need the login credentials of the users account. The user then passes the returned token id and token secret to the application so it can use it. They can also decide to delete the token to revoke access to the application by performing a DELETE request.

Once the token id and secret are known to the application, it can request an access token, which allows the application to perform authenticated requests for a period of time. The service can do this by sending a POST request to the token endpoint with the token id and secret in an `Authorization: Basic` header and a DPoP[34] key. The token endpoint can be found in `/.well-known/openid-configuration`.

With this the access token, requests can be made using the same DPoP key as above and the token. The Solid library *solid-client-authn-core* provides helpers for both DPoP generation and creating functions for authenticated requests.

As the Solid authentication library is written in JavaScript, this service is as well. The Docker image is based on `mu-javascript-template`, which is similar to `mu-python-template`, but for JavaScript services instead of Python.

The service has one endpoint `/update`, which can be POSTed to. If a POST request arrives, the service will recursively query its preconfigured pod and store the data in the database.

The source code of the service is split up across three files

- `app.js`: Contains the entrypoint code, sets up and checks the environment.

- `solid.js`: Contains all code for interacting with the Solid pod, exports the function `getPod`.
- `sparql.js`: Contains all code for writing data to a SPARQL store, exports the function `writeToStore`

5.1. app.js

```
function on_start() {
    dotenv.setup()
    if not all required environment vars are set {
        error()
    }
}

function update() {
    pod_data = solid.getPod()
    sparql.writeToStore(pod_data)
}
```

Listing 19: Pseudo code overview of the file app.js in solid-sync-service.

The first file, `app.js` initializes the environment using `dotenv`[35], which allows the creation of a file named `.env` in the root of the project that contains a set of environment variables for configuring the application. This way, it is also possible to specify the configuration using environment variables for deployment, when there might not be an option to create a `.env` file.

After configuring the environment variables, a check that all required environment variables are present is performed in the function `checkEnv`. If any of these variables are not present, it throws an error during startup. The required variables are:

- `POD_BASE`: The base URI of the Solid pod, also used to find IDP endpoints.
- `POD_NAME`: The name of the pod to synchronize.
- `TOKEN_ID`: The id of the client token.
- `TOKEN_SECRET`: The secret of the client token.

Every time a request on `/update` comes in, the function `update` gets called, which constructs the URI of the Solid pod by concatenating the `POD_BASE` and `POD_NAME`. It also checks if the `POD_BASE` ends with a slash and adds a slash if necessary. With this, it gets the data in the pod using `getPod` and writes it to the database using `writeToStore`.

5.2. solid.js

```
function getPod() {
    result_graph = rdflib.graph()
    visited_uris = []
    uris_to_visit = []

    while uris_to_visit is not empty {
        current_uri = uris_to_visit.remove_one()
        visited_uris.add(current_uri)
        container_data = getContainer(current_uri)
        add container_data to result_graph
        for uri in container_data.find(current_uri, "ldp:contains") {
            if uri not in visited_uris and uri not in uris_to_visit {
                uris_to_visit.add(uri)
            }
        }
    }

    return result_graph
}
```

Listing 20: Pseudo code overview of the file solid.js in solid-sync-service.

In `solid.js`, there is one entry function, namely `getPod`, which performs a depth-first search through the pod and keeps all the results in an `rdflib.graph`. It does this by keeping a list of visited URIs and a list of URIs to visit. The former starts empty, the latter contains just the URI of the pod. It then repeatedly takes the first URI of the URIs to visit, adds it to the list of visited URIs and requests its content using `getContainer`. It then checks the content of the container if it contains more resources by finding all URIs that match `(?currentResourceURI, ldp:contains, ?otherResourceURI)`. It then adds them to the list of URIs to visit if it is not in the list of visited URIs and if it's not already in the list of URIs to visit. It repeats this until the list of URIs to visit is empty.

The function `getContainer` takes a URI and requests the data on that URI using an authenticated fetch created by the `getAuthenticatedFetch` function. If the response is not `200 OK`, it throws an error indicating which status code it got and on which URI. This can happen if a user is not authorized to read a resource, in which case the status code would be `401 Unauthorized`.

The function `getAuthenticatedFetch` will perform the steps described in the Design section, by using the helper functions provided by `solid-client-authn-core`.

5.3. sparql.js

```
function writeToStore(quads) {
    query = "INSERT DATA {"
    for quad in quads {
        query += "GRAPH {} { {} {} {} . }".format(
            sparqlify(quad.graph),
            sparqlify(quad.subject),
            sparqlify(quad.predicate),
            sparqlify(quad.object)
        )
    }
    query += "}"
    query.execute()
}

function sparqlify(node) {
    if node is a named node {
        return helpers.sparqlEscapeUri(node.uri)
    } else {
        if node.type is "string" {
            return helpers.sparqlEscapeString(node.value)
        } else if node.type is "number" {
            return helpers.sparqlEscapeString(node.value)
        }
        ...
    }
}
```

Listing 21: Pseudo code overview of the file sparql.js in solid-sync-service.

The `sparql.js` file contains two functions. The first is `sparqlify`, which takes one rdflib node and transforms it into a string that can be used in a SPARQL query. For this it first checks if it is a named node, in which case it can simply return it as a URI using `sparqlEscapeUri`. If it is a Literal, it has to check which type of literal and use the corresponding `sparqlEscape` function. The second is `writeToStore`, which takes a list of quads and creates and executes a SPARQL query to insert them.

Using this function, a SPARQL query can be created that iterates over each quad in the graph and create a query from those. This is done by creating a different `GRAPH` block for each quad following the format `GRAPH ?g { ?s ?p ?o. }` where `?g` is the URI of the graph, `?s` the subject, `?p` the predicate and `?o` the object. This is done for each quad and they are joined using newlines and wrap them with `INSERT DATA {` and `}`, resulting in a valid SPARQL query.

While this is not a very efficient approach, the JavaScript implementation of rdflib does not provide an alternative approach so this method was chosen.

6. Named Entity Recognition Service

The service that performs the actual named entity recognition is the *named-entity-recognition-service*. This service performs a very basic named entity recognition. If it finds a match, it adds a link from the constituency node to the character to know who it refers to.

This service serves as an example on how complex analyses can be performed in a simple way, once all the required steps are set up. This shown by the fact that the entire service mainly consists of one SPARQL query.

The implementation of this service is very basic. It uses *mu-python-template* and added an endpoint `/annotate` that performs the SPARQL query in Listing 22.

```
PREFIX  ttrpg: <https://w3id.org/TTRpg#>
PREFIX  olia: <http://purl.org/olia/olia.owl#>
PREFIX  itsrdf: <http://www.w3.org/2005/11/its/rdf#>
PREFIX  nif:  <http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core#>
PREFIX  foaf: <http://xmlns.com/foaf/0.1/>

INSERT {
    GRAPH ?g {
        ?node itsrdf:taIdentRef ?character .
    }
}
WHERE {
    GRAPH ?g {
        ?node       nif:posTag   olia:ProperNoun ;
                    nif:isString ?characterName .
        ?character  a            ttrpg:Character ;
                    foaf:name    ?characterName
    }
}
```

Listing 22: SPARQL query to perform the basic named entity recognition.

This query first tries to find all the nodes that have the `nif:posTag` of `olia:ProperNoun`. Then, it finds out which string they represent using `nif:isString`. After that, it finds all `ttrpg:Character`s that have the string as `foaf:name`. Once these nodes and characters are found, a link between them is added using `itsrdf:taIdentRef`.

Chapter 7: Results

Overview

This section will first explain how the application was set up and configured. After that it will show how each step performed by performing some analyses and timings. All timings were run on a laptop with 8GB RAM, 10GB swap and an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz

The source and configuration files of all the services and the system as a whole can be found in GitHub repositories under the MIT license. This way, anyone can contribute to the system or run their own instance.

The main repository for the application is on https://github.com/Robbe7730/app-gelinkt-rollenspelen, this contains the docker-compose setup for the entire stack, as well as configuration files for these services. Each service also has its own repository:

- Sentence Service: https://github.com/Robbe7730/sentence-service
- Constituency Tree Service: https://github.com/Robbe7730/constituency-tree-service
- Named Entity Recognition Service: https://github.com/Robbe7730/named-entity-recognition-service
- Solid Sync Service: https://github.com/Robbe7730/solid-sync-service

1. Configuration

This section will explain the configuration of the services that make up the application. As the entire application is open-source, this section will only give a higher-level overview of each of the configuration parameters.

An overview of all the used services and their interactions can be seen in Figure 15.

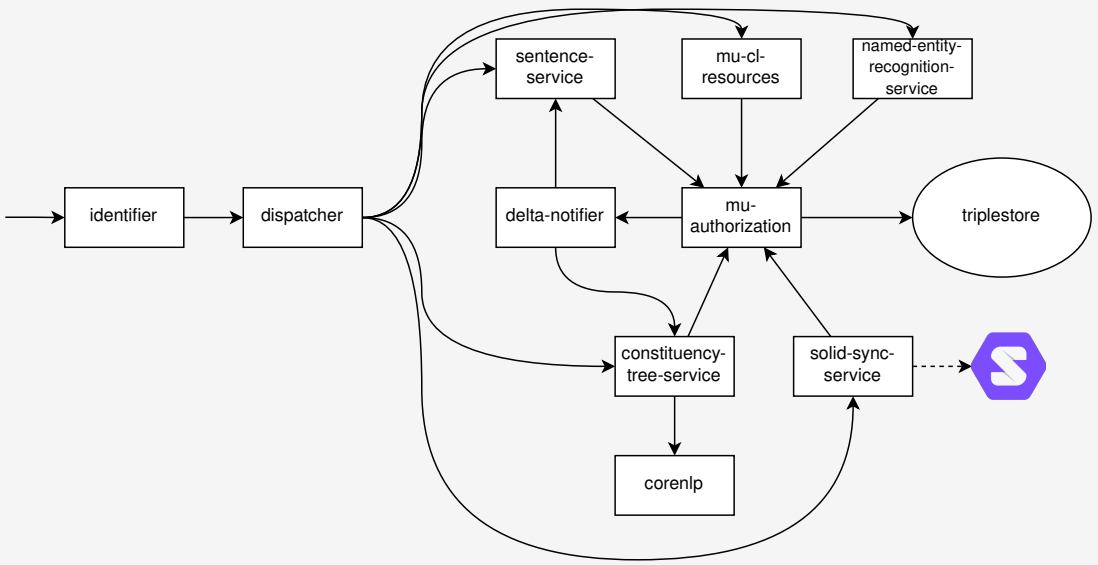

Figure 15: Overview of the application and its services. Each block represents a service and an arrow indicates an interaction from one service to the other. Requests from the frontend enter the system from the identifier.

The middleware services *identifier* and *dispatcher* are in the same place as they are in every Semantic.works stack. To get mu-authorization working, the links to the triplestore have been replaced with links to mu-authorization, which in turn talks to the triplestore. As described earlier, this does not affect the services in any way because mu-authorization uses the same endpoint for SPARQL queries as the triple store. The mu-authorization service also talks to the delta-notifier, who passes its deltas to the sentence-service and the constituency-tree-service. The constituency-tree-service also talks to a separate corenlp container and the solid-sync-service needs access to the Solid pod that it should sync.

The rest of this section will give an overview of how some of the services were configured, more specifically, it will explain the models for mu-cl-resources and the routes for the dispatcher.

1.1. Resources

This configuration defines which models exist and how they are stored in as RDF. There are a total of 5 models defined, namely `text`, `sentence`, `constituency-node`, `character` and `campaign`.

1.1.1. Text

This model defines a piece of `iol:Text`, it has one attribute `value`, which is a *language-string-set* that uses the `rdf:value` predicate. This means that there are multiple string values for this attribute, each with a language tag. The `text` also has a relation to `sentence`, namely which components it has. This is done with the `iol:hasComponent` relation.

1.1.2. Sentence

Similar to `text`, this model defines a `iol:Sentence` with a single string value named `value` using the `rdf:value` predicate. The inverse of the `sentence` relation in `text` is represented here under the name `text`.

1.1.3. Constituency-Node

The most complex model is the `constituency-node` model. It represents each node in the constituency trees using type `nif:String`. It has the following attributes:

- `begin-index` using `nif:beginIndex`: The start of the string in its super-string.
- `end-index` using `nif:endIndex`: The end of the string in its super-string.
- `pos-tag` using `nif:posTag`: The part of speech tag for this node.
- `is-string` using `nif:isString`: The value this node represents.

It also has three relations, namely:

- `sub-string` using `nif:subString`: Relation to zero or more `constituency-node` that are children of this node.
- `super-string` using `nif:superString`: Relation to zero or one `constituency-node` that are the parent of this node.
- `references-character` using `itsrdf:taIdentRef`: Relation to zero or one `character` that this node references, according to the named entity recognition.

1.1.4. Character

The `character` model represents a single `ttrpg:Character`. Currently, it only models the characters name as that is the only value in use. The name of the character is modeled using `foaf:name` and is a string value. As these resource come from an external source, we also tell mu-cl-resources to include the `uri` in the response using the `include-uri` feature. This way, we can link to the character in the Solid pod, even if the data itself comes from the local database.

1.1.5. Campaign

Similar to `character`, `ttrpg:Campaign`s are also modeled. Their model is pretty much identical to `character`, as we only use the name of the campaign.

1.2. Dispatcher

The dispatcher has a configuration that specifies which service a request should be forwarded to.

First, the routes that should be forwarded to mu-cl-resources are specified, otherwise a

frontend would not be able to request this data. These endpoints are:

- `/sentences`
- `/texts`
- `/constituency-nodes`
- `/characters`
- `/campaigns`

After this, every service that has an endpoint that should be public is specified. Each of these routes are forwarded to the corresponding service. These endpoints are:

- `/visualize`
- `/sentence-service`
- `/constituency-tree-service`
- `/named-entity-recognition-service`
- `/solid-sync-service`

All other requests will return a `404 Not Found` with message “Route not found. See config/dispatcher.ex”.

2. Results

This section will give an indication how well each step in the process performed. It will give examples of what went well and what went wrong and why.

2.1. Extracting text from files

Due to the complexity of this step, this was a very manual task that required a lot of fine-tuning and experimenting. Every format came with its own difficulties. While the plain-text formats such as Markdown were easier to read, stripping the special characters from them was more difficult. On the other hand, the Google Documents were harder to get access to, but once this was done the data was pretty much already in plain-text format without special characters.

The fact that the Python library used was unstable and would often crash without any indication why did not help either. The translations were also not always correct. For example, the sentence “Hij komt wel tot het inzicht dat het geluid een val is” is translated as “He does come to the insight that the sound is a fall”, but it would be more correctly translated as “He realizes that the sound is a trap”. The library would also often add or remove whitespace, which would lead to multiple sentences being combined into one because the space after the period was removed.

The mistake that had the most impact on the results of the application was the fact that it sometimes mistranslated names. For example, it would translate the name “Heksina” as “Weksina” most of the time, which makes recognizing this name nearly impossible.

When manually inspecting 100 texts, all of them had their special characters stripped correctly, 76 of them were translated correctly and 24 were translated incorrectly.

2.2. Extracting sentences from text

This step consisted of two SPARQL queries and some Python processing. First, all the `iol:Text`s were extracted from the database, then their values were analyzed using NLTKs `sent_tokenize` function and the resulting sentences were stored back in the database.

The main difficulty was ill-formed sentences. For example, when the sentences did not have a space after their final period, it would see them as one long sentence.

Extracting the 8212 sentences from the 4138 pieces of text took about 1 minute. This is an average speed of around 69 texts per second. Manually inspecting 100 texts showed that all of them were split into sentences correctly.

2.3. Constituency parsing

Constituency parsing was done with CoreNLP, which is a well-known and often used library for this job. Because of this, most of the sentences were parsed correctly. For example, the sentence “You shake them out” gets parsed to the constituency tree in Figure 16.

This constituency tree can be manually verified to be correct, but for larger constituency trees, this process becomes harder as more complex structures can occur.

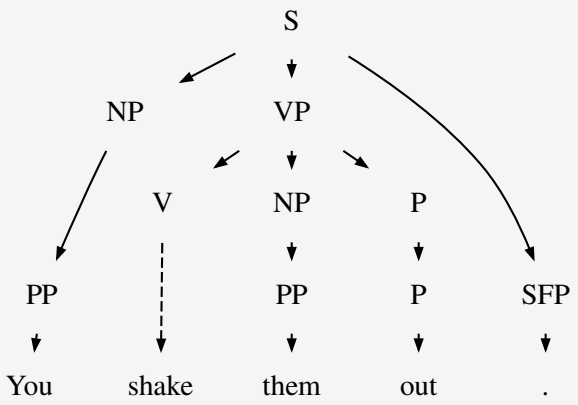

Figure 16: Constituency tree of the sentence "You shake them out."

Where CoreNLP has more trouble is in sentences where punctuation is missing or sentences are ill-formed. For example, the sentence "It's ice cold." is mostly correct, but the correct spelling is "ice-cold", which makes CoreNLP parse that group as a noun phrase instead of an adjective phrase and thus change the interpretation of the sentence. Both constituency trees can be seen in Figure 17.

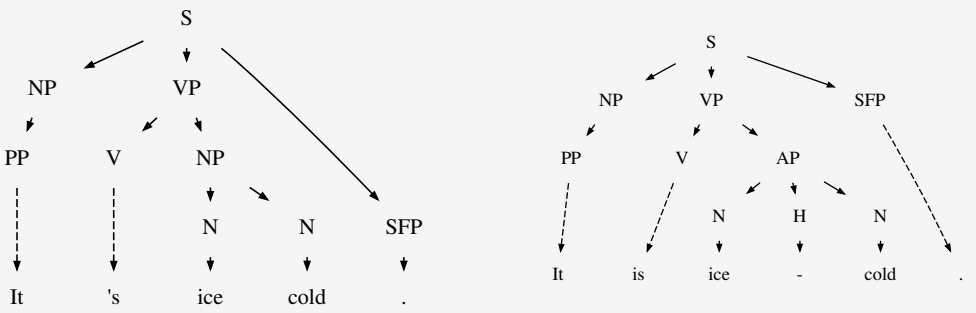

Figure 17: Constituency tree of the incorrect sentence "It's ice cold." and the correct version "It's ice-cold.". The main difference is the AP (adjective phrase) tag instead of the NP (noun phrase) tag.

During testing, the expected time to process all the sentence would be over 5 hours, so a change was made to constituency-tree-service to use 12 threads instead of just 1, which reduced the processing time to 1 hour and 11 minutes and resulted in 228055 nodes. The service failed to process 399 sentences due to various reasons, mainly if they were not sentences at all, such as YouTube links or emoji. This is an average speed of 1.83 sentences and 53.5 nodes per second.

2.4. Recognizing entities

For any correctly tagged constituency tree, the named entity recognition service can correctly extract all names of characters that occur. Even when the constituency tree is not entirely correct but the names are recognized as proper nouns or noun phrases it will still correctly tag the character. In the tests, it was able to extract every occurence of a character from the 3 campaigns that had their data in RDF available and tag it correctly, resulting in 1451 tagged nodes. Mirroring the data from a Solid pod took 52 seconds and running the NER took 8 seconds.

Chapter 8: Conclusion and Future Work

1. Conclusion

This section will draw some conclusions from the results of the steps described above and compare them to what was expected. Finally, it will also evaluate the application as a whole.

1.1. Evaluating in Steps

The first step where the text was extracted from existing files and if needed, translated was the step that performed the worst, mainly due to translation errors. As stated in Subsection 7.2.1, about 24% of the sentences that were originally in Dutch were incorrectly translated to the point where the meaning was changed. This makes the rest of the analysis less accurate, so we will evaluate the next steps in a way that makes this inconsistency less relevant. It should also be noted that this step will be irrelevant in the ideal vision of the project, as the texts would be inserted in the system directly.

The sentence extraction-step was also able to process all 4138 texts in less then 1 minute and a manual inspection found no sentences that were split incorrectly.

The constituency parsing was not able to process each sentence, mainly due to unexpected characters in the sentences. Out of the 8212 sentences, only 399 were not able to be processed. Inspecting the failed sentences revealed that the most common culprit was the use of incorrect symbols, such as unicode or invalid punctuation.

Named entity recognition performed its analysis in less then 10 seconds and was able to find 1451 nodes. It did not miss a single occurrence of a characters name in all the texts, which means that both the constituency parsing and the NER work reliably.

All the steps combined result in a system that works very reliably and produces data in a way that future additions can reuse and build on.

1.2. Evaluating as a whole

The total runtime of the application was 1 hour and 13 minutes for over 4000 texts, which brings the average to around 1 second per piece of text. In a real application, this is a perfectly reasonable time to wait, especially since the data would be analyzed in the background, causing no delay to the user while typing.

The original goal of the project was to go a step further and use the constituency trees in combination with VerbNet[36] to extract events from them. While researching and implementing however, it became clear that even though the idea had potential, it would be harden than expected to implement in time due to the complexity of the idea. The decision was made to focus on only the NER to prove that the system was viable. The idea is described more in Subsection 8.2.1.

Originally, the proof of concept would include a frontend where the user could enter text in an online editor that would send the texts to the backend for analysis and provide a visual indication of the added semantics. This idea was not realized as using an editor that supports RDFa as well as storing the documents in RDF turned out to be more complex than expected.

Even though some steps were not realized, the system definitely shows potential. Every reasoner adds data that is reliable, which leads to a decent result and shows that providing data in a reusable, open and standardized way can provide major benefits to RPG players.

2. Future work

This thesis was also meant to provide the basis of a system that can be expanded on, so the rest of this chapter will describe some examples of services that could be implemented or applications that can make use of the data provided by them.

2.1. Event Extraction With VerbNet

VerbNet[37] is a lexicon containing descriptions of many English verbs and their relation. It groups verbs into syntactic frames, which are a collection of verbs that share both semantic and syntactic features. Every verb class has so-called surface realizations, that indicate how this verb is used. For each constituency node, it indicates what semantic meaning it has to the verb. For example, the verb “approach” belongs to the frame of “escape”, more specifically, to the subclass “escape-51.1-1-2” which also contains the verb “enter”. It has a surface realization of “NP V NP” where the first noun phrase is the theme and the second noun phrase the destination. For example in the sentence “Ronald and Vincent approach the castle”, the noun phrase “Ronald and Vincent” is the theme of the action, the thing in motion and “the castle” is the destination.

With these abstract representation of the verbs, we can extract events from the logs and automatically create a summary of sessions[38] or combine logs from different players into one shared log. The latter of which was originally intended to be part of this thesis, but was cut due to time constraints and unexpected difficulties.

This service could reuse the constituency trees produced by the constituency tree service and add tags to the nodes, just like the named entity recognition service did. Another service could then perform the combination or summarization.

2.2. Live Tagging

In the current state, the applications requires text to be inserted manually before it can be analyzed. In an ideal case, this would be done by having the user type their logs in a tailor-made editor that can talk to the backend automatically. This would allow the user to confirm or deny certain tags, which would reduce the number of incorrect tags significantly.

This editor would also allow the user to store their data as structured documents instead of separate texts. IOLite has classes and predicates that allow the representation of full documents in RDF, which could easily be plugged into the current system. Ideally, the document would be stored in the users Solid pod, to allow them to control who can and cannot access the logs.

2.3. Multiple Campaigns in One World

With this technology, it would be possible for two entirely separate campaigns to take place in the same world. The GMs could use data from both campaigns, which would lead to interesting story elements, where one party may help a village where the people are starving to get more food, but when the next party arrives, there has been a revolution and the whole village is now too industrial.

Using a system as proposed in this thesis would allow both campaigns to use their own set of tools, independently of one another but still be able to share the data.

Bibliography

[1] Y. Wilks and C. Brewster, “Natural Language Processing as a Foundation of the Semantic Web,” *Foundations and Trends® in Web Science*, vol. 1, Jan. 2009, doi: 10.1561/1800000002.

[2] D. Maynard, K. Bontcheva, and I. Augenstein, “Natural Language Processing for the Semantic Web,” *Synthesis Lectures on the Semantic Web: Theory and Technology*, vol. 6, no. 2, pp. 1–194, Dec. 2016, doi: 10.2200/S00741ED1V01Y201611WBE015.

[3] J. L. Martinez-Rodriguez, A. Hogan, and I. Lopez-Arevalo, “Information Extraction Meets the Semantic Web: A Survey,” *Semantic Web*, vol. 11, no. 2, pp. 255–335, Feb. 2020, doi: 10.3233/SW-180333.

[4] S. Hellmann, J. Lehmann, S. Auer, and M. Brümmer, “Integrating NLP Using Linked Data,” in *Advanced Information Systems Engineering*, vol. 7908, C. Salinesi, M. C. Norrie, and Ó. Pastor, Eds. Springer Berlin Heidelberg, 2013, pp. 98–113. doi: 10.1007/978-3-642-41338-4_7.

[5] A. Gangemi, V. Presutti, D. Reforgiato Recupero, A. G. Nuzzolese, F. Draicchio, and M. Mongiovì, “Semantic Web Machine Reading with FRED,” *Semantic Web*, vol. 8, no. 6, pp. 873–893, Aug. 2017, doi: 10.3233/SW-160240.

[6] I. Augenstein, S. Padó, and S. Rudolph, “LODifier: Generating Linked Data from Unstructured Text,” in *The Semantic Web: Research and Applications*, 2012, pp. 210–224. doi: 10.1007/978-3-642-30284-8_21.

[7] K. K. Bowden, J. Wu, S. Oraby, A. Misra, and M. Walker, “SlugNERDS: A Named Entity Recognition Tool for Open Domain Dialogue Systems,” no. arXiv:1805.03784. arXiv, May 09, 2018. Available: http://arxiv.org/abs/1805.03784

[8] “Xml-Dev - RE: [Xml-Dev] Beyond Ontologies.” Available: http://lists.xml.org/archives/xml-dev/200310/msg00269.html

[9] J. D. Gan, “[Rdfweb-Dev] Re: FOAF and MUDs.” Wed May 12 19:24:13 UTC 2004. Available: https://lists.foaf-project.org/pipermail/foaf-dev/2004-May/007369.html

[10] “FOAF Vocabulary Specification.” Available: http://xmlns.com/foaf/spec/

[11] “TTRpg/TTRpg.Ttl at Master · Rwambacq/TTRpg.” Available: https://github.com/rwambacq/TTRpg/blob/master/TTRpg.ttl

[12] “Dungeons & Dragons | Official Home of the World’s Greatest Roleplaying Game.” D&D Official | Dungeons & Dragons. Available: https://dnd.wizards.com//

[13] “Turtle - Terse RDF Triple Language.” Available: https://www.w3.org/TeamSubmission/turtle/

[14] “HTML Standard.” Available: https://html.spec.whatwg.org/

[15] “RDFa.” Available: https://rdfa.info/

[16] “SPARQL 1.1 Query Language.” Available: https://www.w3.org/TR/sparql11-query/

[17] “IOLite Ontology.” Available: http://www.ontologydesignpatterns.org/ont/dul/IOLite.owl#

[18] “Ontology:DOLCE+DnS Ultralite - Odp.” Available: http://ontologydesignpatterns.org/wiki/Ontology:DOLCE%2BDnS_Ultralite

[19] “NIF 2.0 Core Ontology.” Available: https://persistence.uni-leipzig.org/nlp2rdf

/ontologies/nif-core/nif-core.html#d4e948

[20] C. Chiarcos and M. Sukhareva, “OLiA – Ontologies of Linguistic Annotation,” *Semantic Web*, vol. 6, no. 4, pp. 379–386, Aug. 2015, doi: 10.3233/SW-140167.

[21] “ITS 2.0 / RDF Ontology.” Available: https://www.w3.org/2005/11/its/rdf-content/its-rdf.html

[22] “Internationalization Tag Set (ITS) Version 2.0.” Available: https://www.w3.org/TR/its20/

[23] “Project Jupyter.” Available: https://jupyter.org

[24] “HackMD - Collaborative Markdown Knowledge Base.” HackMD. Available: https://hackmd.procore.com

[25] “Obsidian.” Available: https://obsidian.md/

[26] “MediaWiki.” Available: https://www.mediawiki.org/wiki/MediaWiki

[27] “Beautiful Soup: We Called Him Tortoise Because He Taught Us.” Available: https://www.crummy.com/software/BeautifulSoup/#Download

[28] A. Versteden and E. Pauwels, “State-of-the-art Web Applications using Microservices and Linked Data,” Zenodo, Apr. 27, 2016. doi: 10.5281/zenodo.1233427.

[29] “SPARQL 1.1 Query Results JSON Format.” Available: https://www.w3.org/TR/sparql11-results-json/#select-encode-terms

[30] “JSON:API — A Specification for Building APIs in JSON.” Available: https://jsonapi.org/

[31] “Mu-Semtech/Mu-Python-Template: Template for Running Python/Flask Microservices.” Available: https://github.com/mu-semtech/mu-python-template

[32] “NLTK :: Natural Language Toolkit.” Available: https://www.nltk.org/

[33] “CoreNLP.” CoreNLP. Available: https://stanfordnlp.github.io/CoreNLP/

[34] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. Jones, and D. Waite, “OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP),” Internet Engineering Task Force, Internet Draft draft-ietf-oauth-dpop-08, May 2022. Available: https://datatracker.ietf.org/doc/draft-ietf-oauth-dpop

[35] motdotla, “The Worldwide Standard for Securing Environment Variables.” Dotenv. Available: https://dotenv.org/

[36] K. K. Schuler, “VerbNet: A Broad-Coverage, Comprehensive Verb Lexicon,” University of Pennsylvania, 2005. Available: https://repository.upenn.edu/dissertations/AAI3179808

[37] M. Green, O. Hargraves, C. Bonial, J. Chen, L. Clark, and M. Palmer, “VerbNet/OntoNotes-Based Sense Annotation,” in *Handbook of Linguistic Annotation*, N. Ide and J. Pustejovsky, Eds. Springer Netherlands, 2017, pp. 719–735. doi: 10.1007/978-94-024-0881-2_26.

[38] X. Han, T. Lv, Z. Hu, X. Wang, and C. Wang, “Text Summarization Using FrameNet-Based Semantic Graph Model,” *Scientific Programming*, vol. 2016, pp. 1–10, Jan. 2016, doi: 10.1155/2016/5130603.