

Tesla

A SEARCH ENGINE

Team:

Bas METMAN

Vincent VAN MIEGHEM

Supervisors:

dr. ir. Jan HIDDERS (TU Delft)

mr. Eric DANKAART (PwC)

JULY 4, 2014

Contents

Summary	4
1 Introduction	5
2 Assignment & Requirements	6
2.1 Assignment Summary	6
2.2 Requirements	6
2.2.1 Global Requirements	6
2.2.2 Refined Requirements	7
2.2.3 UI Mockups	8
3 Project Methodology	10
3.1 Process	10
3.2 Planning	10
3.3 Tools	11
4 System Design and Implementation	15
4.1 Global System Design	15
4.2 Model	16
4.2.1 SQLite database in Django	17
4.2.2 Solr database	20
4.2.3 Chaining	21
4.3 View	22
4.3.1 Django	22
4.3.2 UI	24
4.4 Back-end	26
4.4.1 Downloader Modules	26
4.4.2 Converter Modules	27
4.4.3 Control	28
5 Testing	29
5.1 Automated Testing	29
5.2 Acceptance Testing	30
5.2.1 Half-way acceptance test	31
5.2.2 Final acceptance test	31
5.3 SIG Feedback	32
5.3.1 Feedback on Duplication	32
5.3.2 Feedback on Unit Complexity	32
5.3.3 Improvements based on feedback	32
6 Future Work & Improvements	35
6.1 Django and Hackstack	35
6.2 Optimalisations	35
6.3 Failure proof	35
6.4 Web crawler	35
6.5 Sorting options	35
Conclusion	36
7 Appendix A: The schema for the Solr database	37

8	Appendix B: Feedback SIG	43
9	Appendix C: Code for chaining algorithm	44
10	Appendix D	47

Summary

Searching for information these days is very important for a lot of professionals. PriceWaterhouseCoopers (PwC) is a multinational professional services network company. As would be expected, professionals at PwC also search several websites daily for specific information to ensure the quality of the service they deliver to their clients.

For this Bachelor project a custom search engine was developed in order to help the PwC professional to find what they are looking for. The system gets newly published data and information from several predefined sources and indexes these locally to ensure speed and independence. In order to develop this system, a scientific approach was used in combination with an incremental, agile development technique called Scrum. Weekly meetings were held with the group, the supervisors and the domain experts.

The system is an aggregation of several components. The first is a Django webserver that takes care of the content delivery to the webbrowser. A second component is a Solr search engine that indexes more than 50.000 documents. A third component is a back-end system that is responsible for gathering new documents from the sources, converting all these to a unified format, controlling all these components to seamlessly work together.

Users can search for documents from overheid.nl, rechtspraak.nl, belastingdienst.nl and loonheffingen.nl. These sources are presented in different categories, according to the selected interest of the user.

The system features a unique feature that is not available in other information delivery systems. ‘Uitspraken’ from rechtspraak.nl are linked together to ensure that the user gets the latest relevant ‘uitspraak’ of a certain case. For this feature, an algorithm had to be designed to find related ‘uitspraken’ and link them dynamically together.

The system was continuously tested using unit tests and user acceptance tests. After every Scrum cycle, a demo was given to the client to ensure the product evolved to their wishes.

A working beta implementation of the complete system has been delivered as a proof of concept and all initially set requirements were fulfilled.

1 Introduction

Searching through information and data in modern days is becoming more and more important. Searching through and for information lies at the core of many services offered by many different companies. The information used by these services is obtained from knowledge and experience from the people working for the company or delivering the service, or has to be gathered from (un)structured data sources. These days, most of the (un)structured data sources are digital sources. This data is structured in a database or is unstructured information from different websites from the World Wide Web.

PriceWaterhouseCoopers (PwC) is a multinational professional services network company. Its services include assurance services (accountant and audit branch), tax advisory and advisory (mainly consulting activities which covers Strategy, Performance Improvement, Transactions Services etc.).

Within PwC, getting the right information for its clients is key. Besides the knowledge and experience of the PwC professionals, a lot of this information comes from the Internet. Information is often gathered from reliable sources like overheid.nl, rechtspraak.nl, belastingdienst.nl, loonheffingen.nl by using Google. Google does this job well, but there is much room for improvement.

A lot of the valuable metadata of these sources is not being used, which results in poor and/or incomplete results in a lot of search cases. Therefore, a need for a custom search engine arises, one that includes the use of metadata and can offer custom features using this metadata for the Tax advisory professionals of PwC.

In this Bachelor Project, a proof of concept of such a search engine has been developed for the Employee Benefits Tax Consultancy (loonbelasting) practice of PwC. Eric Dankaart, who is partner of the Technology group within this Tax Line of Service of PwC NL, will act as engaging partner and supervisor of the ‘client-side’ of the bachelor project.

This document describes the process during the period in which the project took place, as well as the design decisions that were made and implementation of the system. Section 2 describes the assignment and requirements, documenting the assignment analyses to identify requirements. Section 3 describes the project methodology, describing the process by detailing our agile approach and the project planning and tools used. Section 4 outlines the implementation of the system using a top-down strategy. Section 5 covers the testing approach of the software and the results of the user acceptance tests, as well as the results of the code evaluation done by SIG. Finally, in the last section 6, future work is described as well as a conclusion.

2 Assignment & Requirements

In this section the assignment will be summarised, highlighting the key requirements from the client. In the next subsections, a more detailed and defined set of requirements for the system will be described and explained. Also, some early UI mockups can be found in this section.

2.1 Assignment Summary

PwC's professionals currently use a broad amount of sources for the substantiation of their advisory service to their clients. These sources are primarily the digital sources, found reliable and well-known websites on the Internet. The amount of sources is delimited, which makes searching by Google on the Internet inefficient, as a lot of unreliable sources are among the results of Google. Proposed was a web enabled search engine that only searches in a few reliable and legit sources, but searches them much better, taking into account the available metadata. Ideally, the system should also be able to interpret general language queries. The latter is a requirement that later has been decided to be out of scope of this project.

Summarized, the system should be:

- available via a website.
- indexing just a few sources (four).
- fast, Google-like speeds.
- automatically update its contents from the web.

2.2 Requirements

In this subsection the more global and detailed requirements will be described in the following two subsections.

2.2.1 Global Requirements

We distinguished two kinds of requirements from the beginning according to the MoSCoW model. The 'M', the "Must Haves" and the 'S', the "Should Haves". All of the "Must Haves" requirements have to be met by the end of the project in order for the project to be successfully completed.

Must have:

- Search function to search in sources:
 1. overheid.nl
 2. belastingdienst.nl
 3. rechtspraak.nl
 4. loonheffing.nl
- Search queries should be highlighted in the results.
- Results should be ordered based on relevance.
- Results should be sortable based on the publication date.
- A preview of the resulting documents should be available.
- A link to the original document should be available.

- The preview of the results should be uniform.

The “Should Haves” are defined as being, “These features should be implemented for the project to achieve the ultimate.”

A few of these got implemented, but the third and the fourth requirements have been decided to be out of scope for this project and will not be fulfilled.

Should have:

1. Linked ‘uitspraken’ from courts.
2. Linked legislation.
3. Natural language recognition.
4. Direct answers to queried questions.

2.2.2 Refined Requirements

From the set of global requirements, we defined the set of requirements that would help to structure features in SCRUM sprints in a later stage of the project. We defined them in a simplified version of User Stories used within de BDD methodology. We differentiate two types of requirements, requirements for the user, that define the usability of the system and the requirements for the underlying system, that define the performance of the system.

User Requirements

The user’s UI can roughly be divided in three parts. The Search page, the Result page and the Preview page. For each part of the system a set of user requirements are described below.

Search page.

A user should be able to:

- get to the search page from within their browser.
- enter a search query.
- select relevant categories or sources.
- distinguish search queries
- use logical operators in their search queries

Results page.

A user should be able to:

- get ordered results.
- to click on a result and get a preview of the document.
- sort results based on the publication date of the document.
- re-enter a search query and use all options available on the search page.
- receive results that are linked to each other.
- find the query in the results.

Preview page:

A user should be able to:

- get a preview of the document in uniform format.
- click on a link for the original document.

System requirements

The system requirements should form a guideline of the features and their performance that have to be implemented.

Below are the requirements of the back-end system.

The system should be able to:

- autonomously run 24/7/365.
- get new updates from sources every 24 hours.
- convert 4 different kinds of format to a uniform format.
- convert different sources into a uniform index scheme.
- index and handle 50.000 (and counting).
- Show results in less than 10 seconds.
- Find related ‘uitspraken’ and link them to each other.

Based on the requirements, sprints have been planned. The next section, 3.2 Planning, will go into further detail.

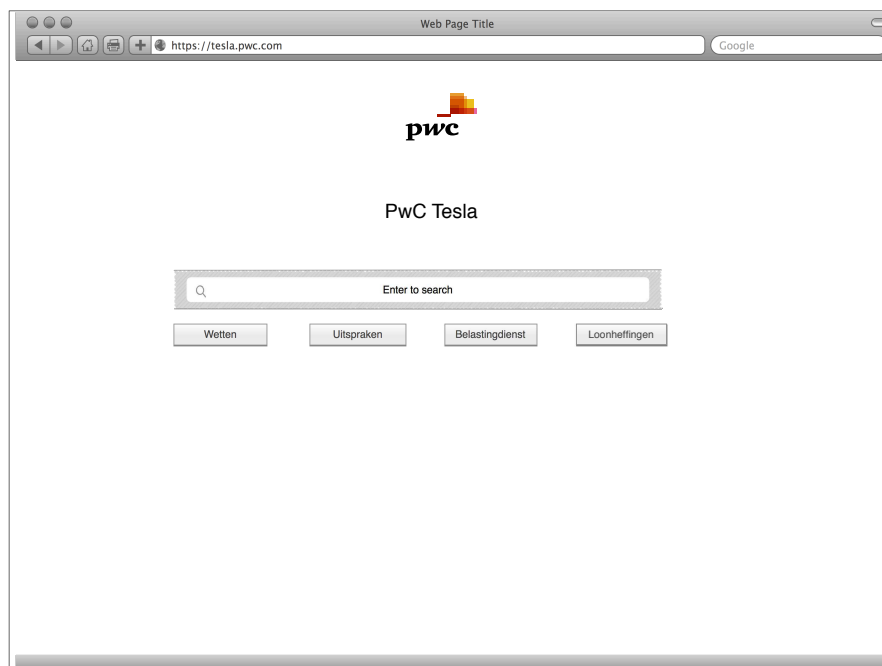
2.2.3 UI Mockups

To get early feedback from the client, UI Mockups were created. Mockups would help to get a clear straighten the vision and UI goal of the system. Usability was key, the system should be very easy to understand for inexperienced computer users. Below, we made three mockups of the system, the Search Page, Results Page and the Preview Page.

Search Page

No distractions, buttons to show or hide categories. Hit the Enter-key to search.

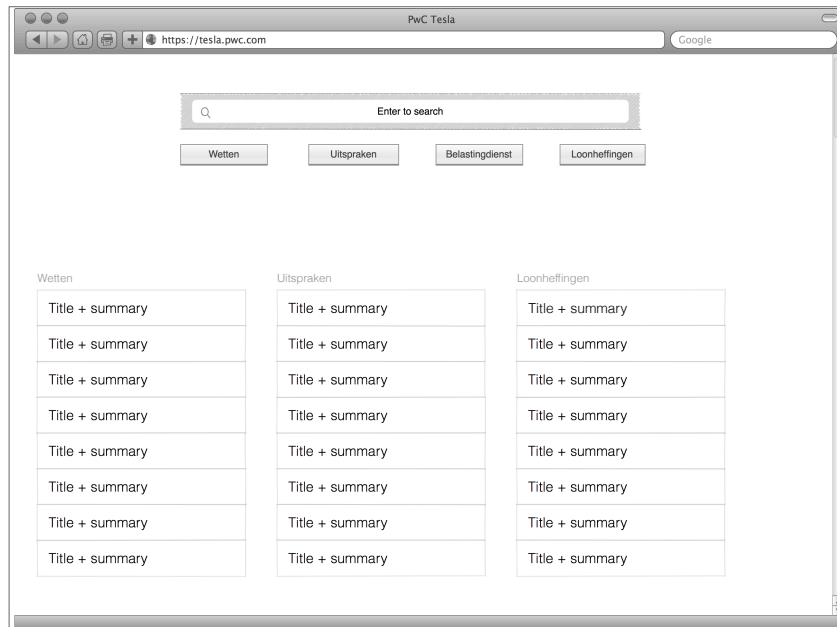
Figure 1: Search Page



Results Page

Again, no distraction, buttons to show or hide categories, ability to re-enter a search query. Results are shown in their category column. Each entry has a Tile and a summary of the document, if available.

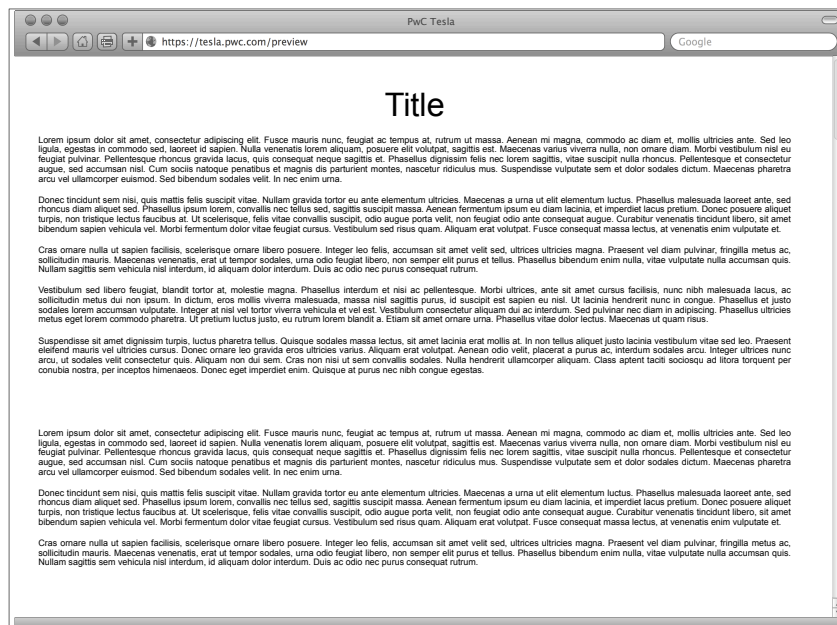
Figure 2: Results Page



Preview Page

If the user clicks on an entry, the user will be redirected to a page where the according document is shown.

Figure 3: Preview Page



3 Project Methodology

This section will focus on the methodology of the project, giving insight in how this project was put together. Firstly, the strategy of the process will be discussed. Furthermore, the planning will be explained. Lastly, the tools used in this process are detailed in the last section.

3.1 Process

The most important aspect of the process of a project is communication. For work to be done smoothly and effectively, communication to all parties has to be maintained. In this case, we had to stay in contact with three parties: the supervisor from the client, the supervisor from TU Delft and of course the team itself.

We maintained communication with the client by working one or two days at the office, mostly when we wanted to demo our product or when we wanted to know what the client exactly wanted or didn't want. When we were there, we also spoke with other employees working at the same or other departments, discussing features of the product and giving demo's whenever we could.

We maintained communication with the supervisor from TU Delft by e-mail, sending a report at the end of every week with the progress of that week. Furthermore, we scheduled a meeting with the supervisor from TU Delft at around the half of the project, to discuss our progress and our findings, and where we could ask for advice on some problems we were having.

On days that we were not at the office, we worked from home. We worked together with Skype, constantly talking to each other. Often enough we were pair-programming, using Skype's 'screen share' function.

These communications could be done effectively because we worked in a small team; there were only two people in our development team. This small size also helped to maintain constant communication, which can be used effectively for agile development. This, and the reason we always had to have a demo available to show to the client, led us to the use of Scrum-inspired development cycles.

Every development cycle was one week. Every week, we decided what functionalities the product should have at the end of the week. When these functionalities were not met or met earlier than expected, we discussed what changes should be made in the development cycle. On top of weekly discussions about the development cycles, we divided these tasks between us every day before we got to work. After a break in the afternoon we discussed our progress, and at the end of the day we reviewed what had been done and should be done the next day. This way, we made sure that we always knew exactly how the development cycles progressed, allowing for informed decisions to be made when implementation did not go as planned.

3.2 Planning

As mentioned in the last section, most of the planning was done over the phone, updating the progress continuously and adjusting goals and planning accordingly. But, we did make an overall planning, detailing what functionality should be worked on and when. This planning is shown here.

Week	Fase	
1	Orientation	Define user requirements
2	Orientation	Analyse requirements and read technology documentation
3	Implementation	Setup of basic functionality
4	Implementation	Setup of basic functionality
5	Implementation	Build chaining functionality
6	Implementation	Setup general language functionality
7	Implementation	Build general language functionaliteit
8	Implementation	Finish general language functionaliteit
9	Testing	User acceptance testing
10	Presentation	Presentation and completion of product

This planning was made assuming that the basic functionality would be fairly easy to implement. As it turned out, the basic searching functionality was indeed according to plans, finished fairly soon. What we did not expect though, was that the sources which we had to include in the search engine took a lot of time to process and index. Also, every source was made in a different format, which led to the slow implementation of new sources. So, we decided to adjust the planning after the second week. The updated planning can be seen here.

Week	Fase	
1	Orientation	Define user requirements
2	Orientation	Analyse requirements and read technology documentation
3	Implementation	Setup of basic functionality
4	Implementation	Finish basic functionality
5	Implementation	Setup indexing and processing of sources
6	Implementation	Finish indexing and processing of sources
7	Implementation	Setup test suite and advanced functions
8	Implementation	Build chaining functionality
9	Testing	User acceptance testing
10	Presentation	Presentation and completion of product

As can be seen, the basic functionality was planned to be done in two weeks. The processing of the sources also took two weeks; long, but necessary for a functional implementation. After finishing this, we decided to set up the test suite for checking if the processing and indexation process were done correctly. Several advanced functions were implemented too, like highlighting results and several visual improvements. The chaining functionality was an important requirement of the client, and we wanted to ensure this worked correctly. Finally, user acceptance testing was done to ensure that the client would be satisfied with the product.

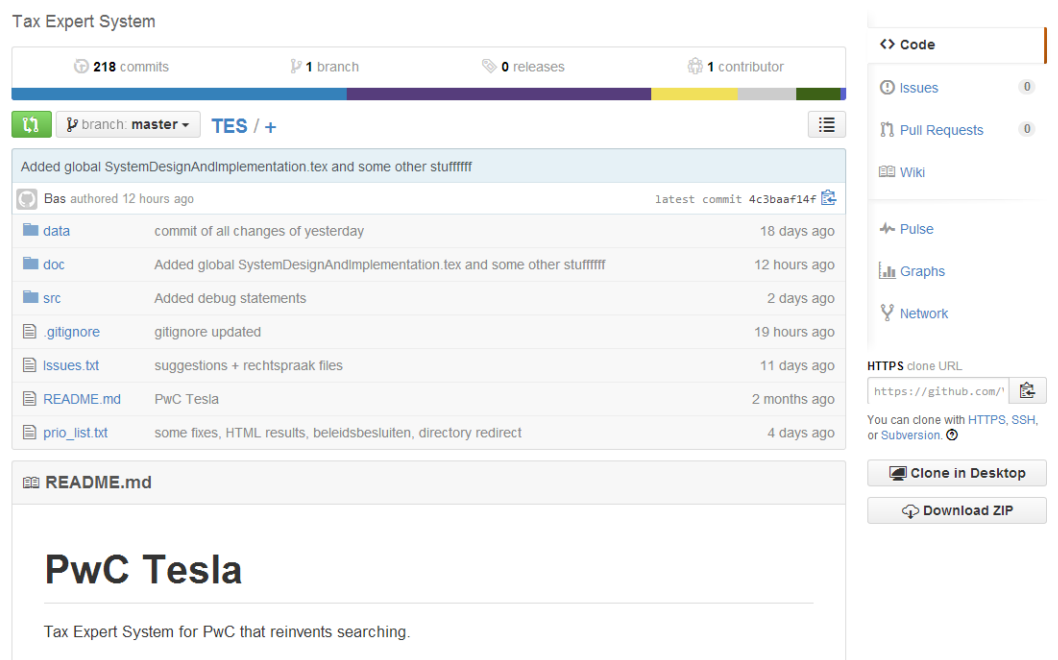
3.3 Tools

An essential part of the project are the tools used for developing the project. In this project, we used only open source tools to keep the costs of the project down. In the rest of this section, a non-exhaustive list of tools will be given with information about every tool.

Github

Github is a web-based hosting service which offers repositories to share code. Github is the largest code host in the world, and we have much experience with using it. The reason why we chose Github is because the ease of access and how easy it is to share code, without having to worry about overwriting code that has been edited by another person. When errors are made, reverting the code to an earlier state is also easily done. This, in addition with open-source tools to access the repository made Github a good choice.

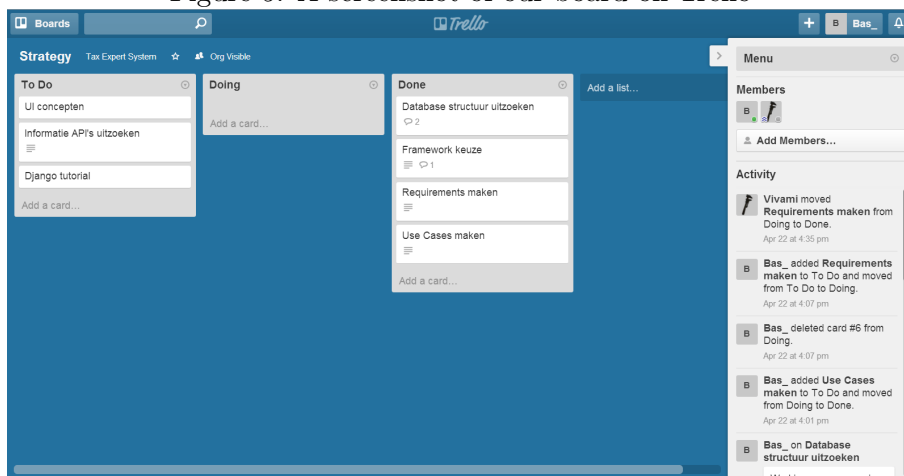
Figure 4: A screenshot of our repository on github.com



Trello

Trello is a web-based project-management application. It allows for creating multiple boards per project, for example for planning, requirements or software features. At the start of the project, we tried to keep track of the different tasks of each development cycle that should be done, have been done and that we were currently doing. This proved redundant as every task was communicated in abundance, and there was no need to write this down. So, Trello was only used in the research phase of the project, in the first two weeks and was abandoned soon after. For some requirements and bugs we had to keep track of, we made a small text document in the github repository which we updated daily.

Figure 5: A screenshot of our board on Trello



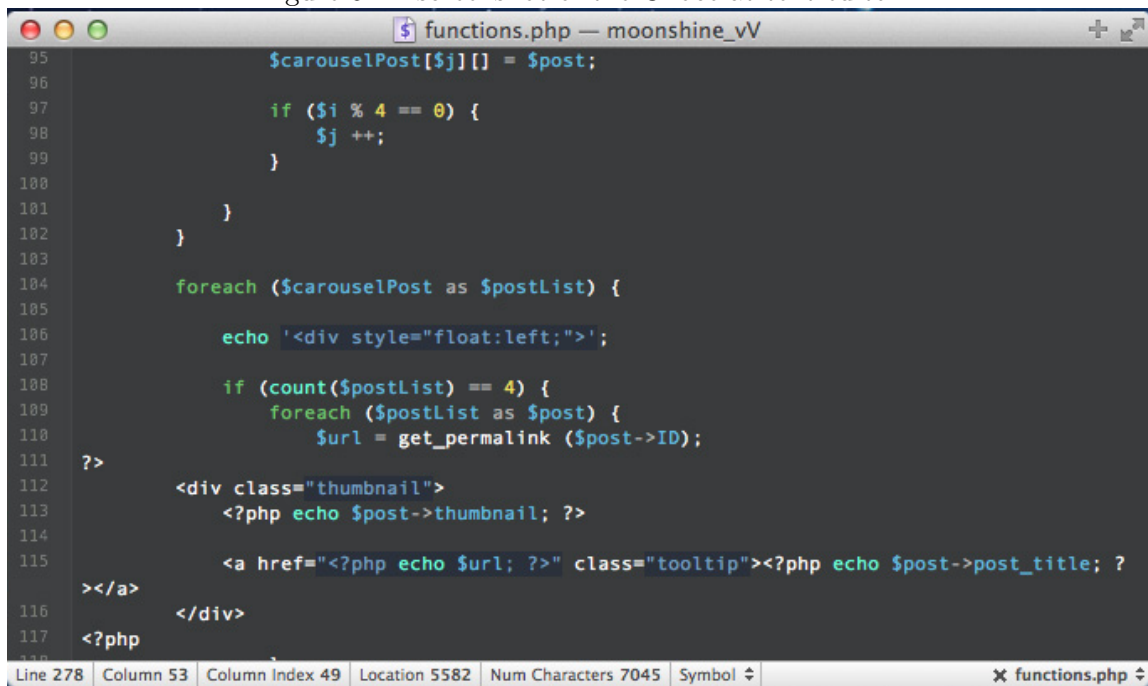
Skype

Skype has been the most vital tool during the development process. Skype is software that allows VOIP, Voice Over IP. It allowed for instant communication and we collaborated on code hours on end while communicating problems and solutions over Skype.

Editors

During the development process we used different editors to edit code. This can be attributed to the fact that the code of the project was written in multiple different languages, for instance Python (Django), Javascript (JQuery) and even Java (Solr). This, plus the fact that works was being done multiplatform on both a Mac and a Windows computer, led to the use of a combination of text editors and IDE's. A text editor or IDE which could handle multiple types of files was thus useful and necessary. So, on the Mac platform the text editor Chocolat was used, while on the Windows platform a combination of the Idle IDE (Python) and Notepad++ was used.

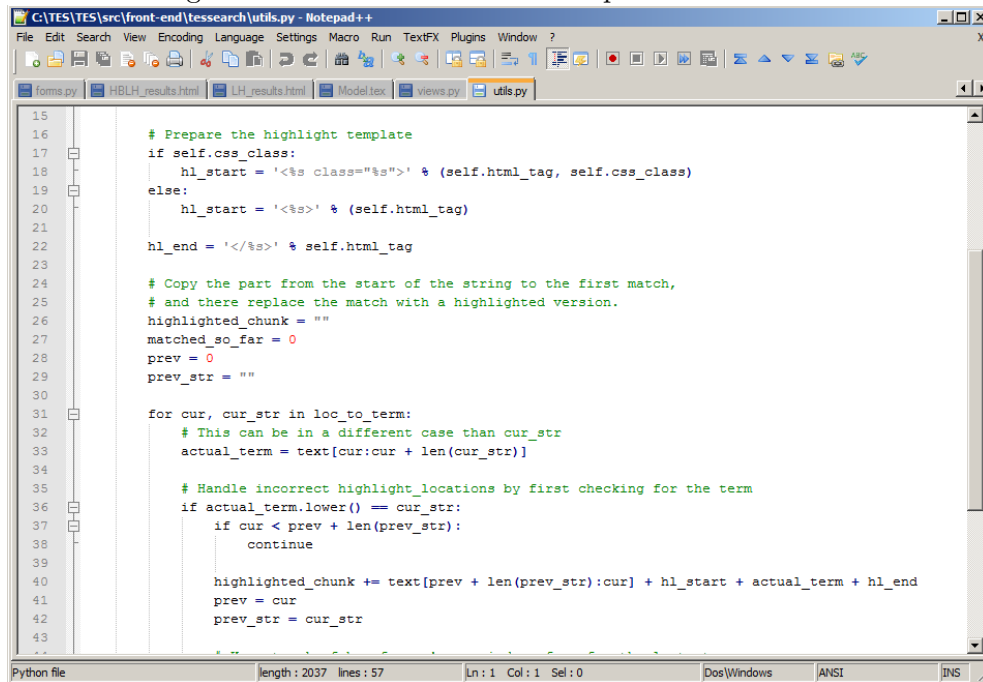
Figure 6: A screenshot of the Chocolat text editor



```
95     $carouselPost[$j][] = $post;
96
97     if ($i % 4 == 0) {
98         $j++;
99     }
100
101 }
102 }
103
104 foreach ($carouselPost as $postList) {
105     echo '<div style="float:left;">';
106
107     if (count($postList) == 4) {
108         foreach ($postList as $post) {
109             $url = get_permalink ($post->ID);
110
111 ?>
112 <div class="thumbnail">
113     <?php echo $post->thumbnail; ?>
114
115     <a href="<?php echo $url; ?>" class="tooltip"><?php echo $post->post_title; ?
116 ></a>
117     </div>
118 <?php
```

Line 278 | Column 53 | Column Index 49 | Location 5582 | Num Characters 7045 | Symbol ↕ | functions.php ↕

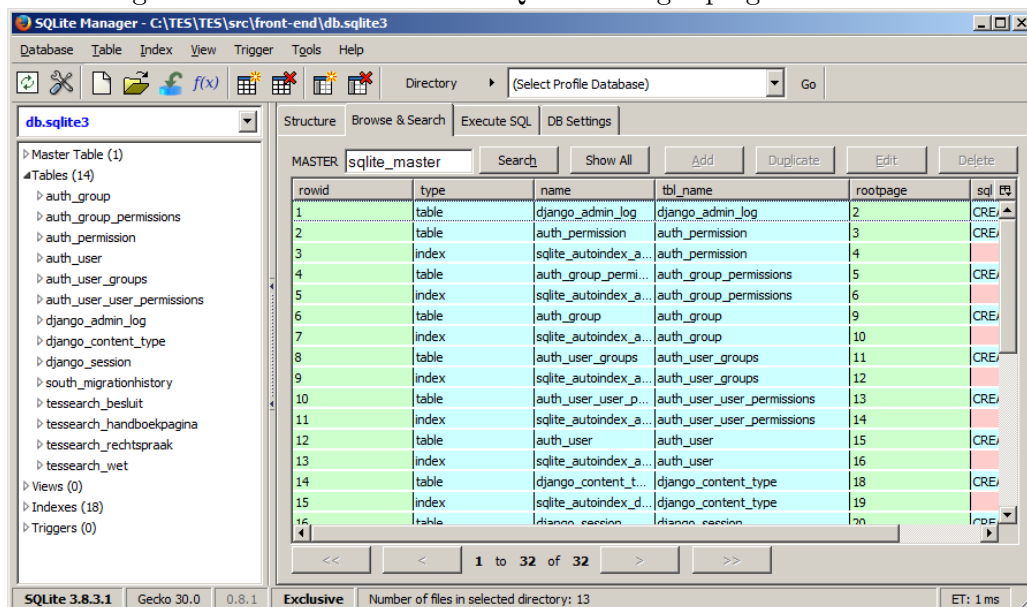
Figure 7: A screenshot of the Notepad++ text editor



SQLite manager

The last tool to be discussed is the SQLite Manager for Firefox. This software is a plug-in for Firefox, allowing to easily see what tables a database has and see what values the field have. As in SQLite the database is only a single file, this plug-in provides ease of access as for setup the file only has to be selected and the database will be loaded. This tool provided a lot of insight in the database during debugging.

Figure 8: A screenshot of the SQLite manager plug-in for Firefox



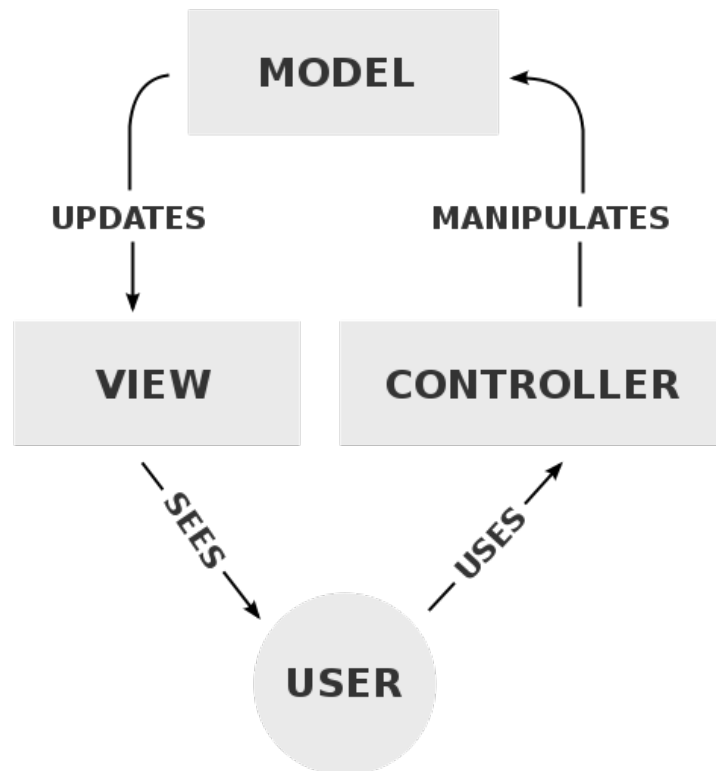
4 System Design and Implementation

In this section, the structure, design and implementation of the final system will be discussed. Specifically, the engineering decisions in this process will be highlighted. First, the global structure and design will be discussed, whereafter the individual components will be discussed. This way, the top-down design strategy that was used during the design will be continued in this section and thus will provide the most insight in the engineering decisions.

4.1 Global System Design

As is usual for Web applications, the system uses the software architectural pattern Model-View-Controller (MVC). This pattern divides the application into three interconnected parts, which each have a different way to handle information. The Model component consists of application data, and contains the used databases. The View presents the information, extracted from the model, to the user and can also accept input commands from the user to be used by the Controller. The last component is the Controller, which further processes the input from the user and communicates these commands to the Model. A diagram for the MVC can be seen in Figure 9.

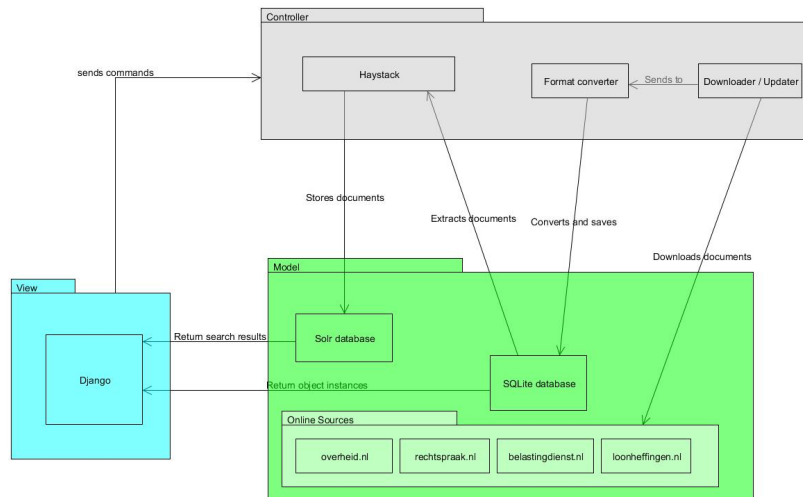
Figure 9: Model-View-Controller



Following this software architectural pattern, a diagram was made for the system architecture. This can be seen in figure 10.

As can be seen, the View component of the system is entirely made in Django. This makes sense, as Django runs on the server and is used to display the HTML-pages used by the user. The MVC pattern is followed closely, as the View can not directly interact with the Model, but has to send commands to the Controller first, where they will be processed further.

Figure 10: Model-View-Controller



The Controller component also adheres to the pattern. It processes commands and can edit the Model. It consists itself of three major components: the downloader/updater, the format converter and Haystack. The downloader/updater downloads documents from the external online sources that have to be index, as stated in the requirements. This component is also in charge of keeping the documents up to date, regularly downloading updates to old documents and/or new documents. These documents all have their own format however, which is why all documents have to be converted by the format convert, which converts the documents in such a way that they can be stored in our own SQLite database in the Model component. After the conversion, the Haystack software can then retrieve these documents from the local database and index them in Solr.

The Model is already been explained for the most part, but one aspect may stand out: the fact that there are two databases used. The reason for this is simple: speed. Documents can be search through faster in Solr, which is why Solr is used to return search results. The Django SQLite database is used for fast retrieval of documents, but never for searching, as this would take too long.

In the next sections, the individual components will be explained in more detail.

4.2 Model

As is common in the Model part of the Model-View-Controller design architecture, the Model part of the system should only be used for storing data. The Controller part updates this data, whereas the View then will use the updated data. Considering all this, we use a database to for the Model part of our system.

For this project, we want to process and index several sources. We want this for two reasons. The first reason is speed. When we store our documents locally, the operation time of a search becomes much faster than if we have to access several online databases for one query. The second reason is that we want to control the how the data looks like. This can be achieved by preprocessing downloaded documents and storing this in our database. The main documents we

want to index in our database are: laws ¹, jurisdiction ², payroll taxes handbook ³ and tax rulings ⁴.

Indexing different documents from different sources is difficult in the way that they are all written in their own format. Furthermore, to download the appropriate documents takes knowledge of the API of the open data database of each source. To this end, we processed downloaded documents and stored them in our own format. As mentioned earlier, a second bottleneck is speed. To reduce the amount of time it takes to return results of a search query, we decided to use Solr ⁵. Solr is an open source search platform, specifically designed to work for web applications. Even for many documents, it can return search results very fast.

However, for data to be indexed in Solr, we have to convert it in a special format recognized by Solr. Because the product used the Django environment, we decided to use Haystack ⁶. Haystack allows for easy communication and indexing between a local Django database and a Solr database. Now, for the Solr database to be filled with documents, only the Django database needs to be filled and Haystack will automatically update the Solr database. We decided to use a SQLite ⁷ database for Django, because this is supported as a standard in Django and also integrates well with Haystack.

In summary, the product uses two databases: a SQLite database in Django and a Solr database. The SQLite database is used to store processed documents and as a reference for Django, whereas the Solr database is used for quickly returning answers to search queries. The next two sections explain how the two databases are designed.

4.2.1 SQLite database in Django

The SQLite database in Django has four different tables, one for each type of document to be stored. These models, as they are called in Django, can not only be used for retrieving data by search queries, but can also have methods which can be used on extracted data from the database. For instance, our **Wet** model has a method **truncated_content**, which extracts the **Inhoud** field from a **Wet** document, and truncates it to 400 characters. This can be used by the View part of the system, the HTML pages used by Django, to show a preview of the contents of a document. The tables/models in the SQLite database are explained in detail in the remainder of this section.

¹<http://koop.overheid.nl/producten/basis-wetten-bestand>

²<http://www.rechtspraak.nl/Uitspraken-en-Registers/Uitspraken/Pages/default.aspx>

³http://www.belastingdienst.nl/wps/wcm/connect/bldcontentnl/belastingdienst/zakelijk/personeel_en_loon/handboek_loonheffing

⁴<http://www.loonheffing.nl/Loonheffing-MvM.htm>

⁵<http://lucene.apache.org/solr/>

⁶<http://haystacksearch.org>

⁷<http://www.sqlite.org>

Wet model

Name	Type	Usage
BWBId	Char Field	An ID for the document given by the source
DatumLaatsteWijziging	Text Field	Last modified on
VervalDatum	Text Field	Date when the document is not legally valid anymore
OfficiëleTitel	Text Field	The official title of the document
CiteerTitel	Text Field	The title used to cite the document
Status	Text Field	The status of the document
Titel	Text Field	The informal title
InwerkingsTredingsDatum	Text Field	Date when doc became legally valid
Afkorting	Text Field	The abbreviation from the document
RegelingInfo	Text Field	Extra information about the document
Bron	Text Field	An HTML link to the source of the document
Inhoud	Text Field	The contents of the document
SorteerDatum	DateTime field	Date used by Django to sort the document
__unicode__	Method	Returns a String representation of the object.

The **Wet** model/table is used to store legal law documents. Most of the fields self-explanatory, but some fields need further explanation. First off, the only DateTime field is the **SorteerDatum** field. There are other 'Datum' fields, but these are only necessary to show information in the View, not for sorting methods. The **SorteerDatum** field however is used to sort the results and is thus a DateTimeField.

Secondly, it can be noticed that no primary key is declared. An obvious contender would be the **BWBId** field, as it already an ID for the document. But, in some cases and in some sources, this key is not unique and thus can not be used as primary key. So, a primary key is implicitly declared by Django and is automatically created.

A final remark is that the `__unicode__` method must be included for every model declared in Django. Else, Python can't return a String representation of an object instance.

Rechtspraak model

Name	Type	Usage
ECLI	Char Field	ID for the document given by the source
Inhoudsindicatie	Text Field	Indication of what the contents could be
DatumLaatsteWijziging	Text Field	Last modified on
CiteerTitel	Text Field	The title used to cite the document
Maker	Text Field	The institution that made this ruling
Uitgever	Text Field	The institution that published this ruling
Taal	Text Field	The language of the ruling
Vervangt	Text Field	ECLI ID of the ruling that this ruling overrules
Uitgavedatum	Text Field	Date of publishing
Uitspraakdatum	Text Field	The language of the ruling
Zaaknummer	Text Field	The number of the case discussed
Type	Text Field	Type of ruling
Procedure	Text Field	The procedure of the ruling
Rechtsgebied	Text Field	The legal area of the ruling
Titel	Text Field	The title of the document
Bron	Text Field	An HTML link to the source of the document
Inhoud	Text Field	The contents of the document
RelatieType	Text Field	Relationship with an other relation
RelatieECLI	Text Field	ECLI of the other rulings this ruling has a relation with
SorteerDatum	DateTime field	Date used by Django to sort the document
__unicode__	Method	Returns a String representation of the object.
truncated_content	Method	Returns the Inhoud field truncated to 400 characters
get_instance	Method	Returns what instance has made the ruling
get_related_documents	Method	Returns the related Rechtspraak objects

The **Rechtspraak** model/table is used to store legal rulings. Most of the fields here are again metadata for the document. The most important fields are **RelatieType** and **RelatieECLI**. The **RelatieECLI** contains a single or multiple ECLI ID's for other **Rechtspraak** documents. This field can be used by the **get_related_documents** method to retrieve all related **Rechtspraak** objects. The **RelatieType** field indicates what kind of relation it has with the other documents. The relations could also be represented with foreign keys, but with the ability to use methods this would only lead to redundancy in the database. Furthermore, the **SorteerDatum** field is again used for sorting the results. Lastly, the **get_instance** field is used for extracting the instance, something that is not explicitly defined in the metadata.

Handboekpagina model

Name	Type	Usage
HBLH	Char Field, max length = 30	An ID for the document given by the source
Jaar	Text Field	The year this document was published
Titel	Text Field	The title of the document
Bron	Text Field	An HTML link to the source of the document
Inhoud	Text Field	The contents of the document
SorteerDatum	DateTime field	Date used by Django to sort the document
__unicode__	Method	Returns a String representation of the object.
truncated_content	Method	Returns the Inhoud field truncated to 400 characters

The **Handboekpagina** model/table is used to store pages of the official tax handbook. All fields are self-explanatory.

Besluit model

Name	Type	Usage
HBLH	Char Field, max length = 30	An ID for the document given by the source
Jaar	Text Field	The year this document was published
Gewijzigd	Text Field	The date this document was edited
Titel	Text Field	The title of the document
Bron	Text Field	An HTML link to the source of the document
Inhoud	Text Field	The contents of the document
SorteerDatum	DateTime field	Date used by Django to sort the document
__unicode__	Method	Returns a String representation of the object.
truncated_content	Method	Returns the Inhoud field truncated to 400 characters

The **Handboekpagina** model/table is used to store pages of the official tax handbook. All fields are self-explanatory.

4.2.2 Solr database

Solr is a search platform that can search through a customisable database. This database has to be configured in such a way that all the fields that exist in the Django database can be indexed and stored. Furthermore, all the fields have to be searchable. To achieve this in Solr, a so-called schema has to be written. This schema can also be generated with Haystack, which will automatically configure Solr to work with the Django models. Therefore, we used Haystack to generate such a schema but later edited it manually to provide partial search on all different fields. A detailed explanation will be given with snapshots of this schema.

Schema.xml codesnippet 1.

```

1      ...
2      <field name="RelatieECLI" type="text_en" indexed="true"
3      stored="true" multiValued="false" />
4
5      <field name="BesluitID" type="text_en" indexed="true"
6      stored="true" multiValued="false" />
7
8      <field name="Gewijzigd" type="text_en" indexed="true"
9      stored="true" multiValued="false" />
10
11     <field name="SorteerDatum" type="date" indexed="true"
12     stored="true" multiValued="false" />
13
14     </fields>
15     ...

```

This code snippet shows how the different fields are stored in the schema. Almost every field is of type text_en, and is indexed so it is searchable. The only exception is the SorteerDatum field, which is of type date, which can store a date object.

Schema.xml codesnippet 2.

```

1      ...
      <!-- field to use to determine and enforce document uniqueness. -->
3      <uniqueKey>id</uniqueKey>

5      <!-- field for the QueryParser to use when an explicit fieldname is absent
      <defaultSearchField>text</defaultSearchField>

7      <!-- for letting solr search in every field, not just text-->
9      <!-- for Wet model -->
      <copyField source="Status" dest="text"/>
11     <copyField source="CiteerTitel" dest="text"/>
      <copyField source="DatumLaatsteWijziging" dest="text"/>
13     ...

```

In this part of the schema a few things can be seen. First, an id field is defined for automatically generating a primary key for Solr to use. Secondly, a defaultSearchField named 'text' is defined, which is used by the QueryParser to search. Finally, every field is copied into the text field, so that every field can be search in the Solr. The full schema can be found in Appendix A 7 .

4.2.3 Chaining

A major functionality that was defined as requirement by the client was the 'chaining' of Rechtspraak documenten. Simply put, a Rechtspraak document contains a certain ruling. This ruling can be overruled by a higher instance, as well as overrule another ruling. This can be repeated for any number of documents. As a result, a 'chain' of documents can be represented as a **tree data structure** with any number of nodes.

To collect all relations of a ruling to other rulings, basically, the tree has to collected and has to be flattened. Flattening entails retrieving all nodes from a tree. This is often done via the root of a tree; when the root is found, the flattening is easily done. Unfortunately, our system does not have this luxury. When the relations of a ruling have to be found, it is a random leaf on a random depth. To retrieve the root is not an easy task, because the tree may have two or more nodes at the lowest depth, so it does not have a specific root to retrieve.

So, to achieve this, a custom algorithm had to be made. The main idea of the algorithm is: for every relation of a ruling, recursively find all relations of the related rulings, and return the id's of these rulings in a set. To do this, a set of already found rulings should be tracked, to prevent deadlocks and unnecessary look-ups. An additional detail to be mentioned is that this algorithm is not run just-in-time, but run once on every document in the database, containing more than fifty thousand documents and then saving the found relations in a field of the ruling object. So, when the relations of a certain ruling have been found, the relations of the rest of the rulings are also found, and can be written in their respective fields immediately. These documents can then be skipped, making the algorithm more efficient.

This section will further detail the pseudo code for this algorithm. The full algorithm can be found in Appendix C 9.

Chaining algorithm pseudocode.

```
1 procedure get_related_rechtspraak(Ruling doc, set result){
3     dif = difference(result, doc.relations);
5     if dif is empty{
6         return result;
7     }
8     else {
9         for every relation in dif{
10             result is result + get_related_rechtspraak(relation, result + doc);
11         }
12     }
13     return result;
15 }
```

At instantiation, there are two arguments given for the algorithm: the document of which the relations have to be found and the result set of already found relations. This result set is empty when first calling the algorithm. The algorithm first checks the difference between the already found relations and the relations of the document. if there is no difference, all the relations for this document have already been found and the found resultset is simply returned. Otherwise, for every relation that is not yet found, the algorithm is called recursively on that relation with the original resultset plus the document, because it is now searched. Eventually, every related document is found.

This algorithm is called for every document in the database to find all related documents. The full implementation of this, and the actual implementation of the algorithm, can also be found in Appendix C 9 .

4.3 View

The View in the MVC-model is the part that is responsible for representing the data processed by the Controller and the Model. It also defines the way the users interact with the system, which makes the View a crucial part in the usability of the system.

The following subsections describe structure of the View in the system. Section 4.3.1 will explain the Django Framework set up. Section 4.3.2 will describe the structure of the GUI.

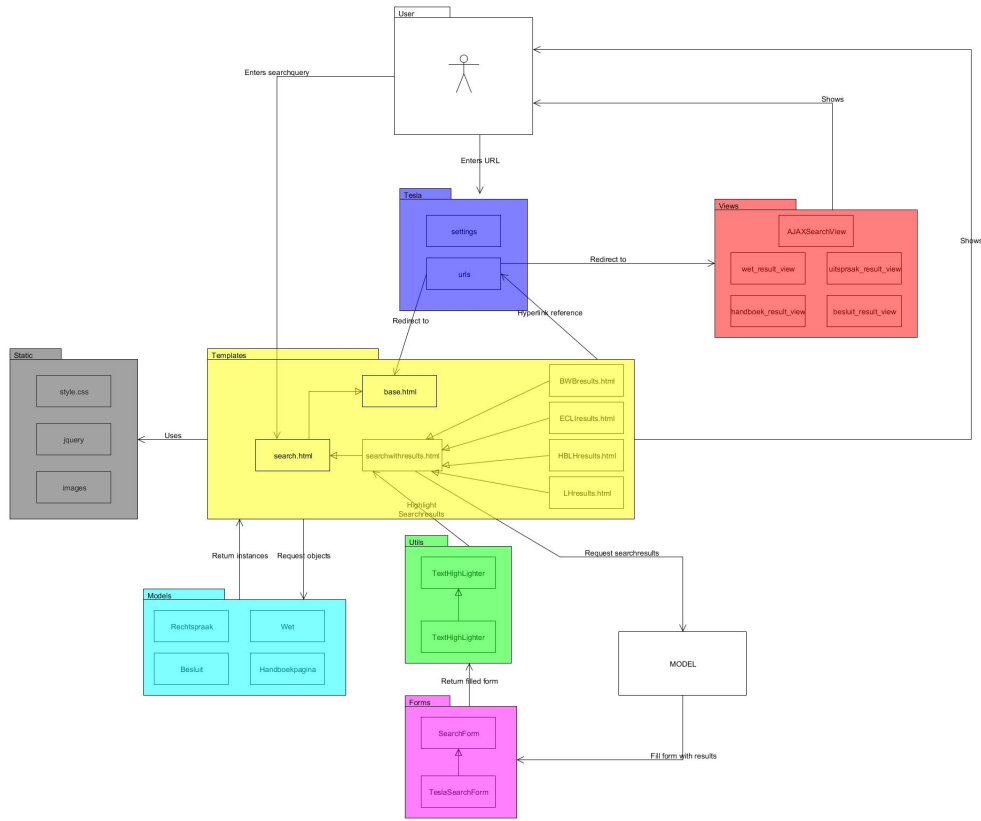
4.3.1 Django

Django ⁸ is a Python Web framework which is designed for rapid development for web application. Regarding the short development time for this application, the Django framework is very suitable. Django comes with an object-relational mapper in which the database layout can be described in Python code, which is very useful in this database-oriented application. The code written for the application in Django is spread out in individual Python files. To make sense of all these pieces of code, this section will explain the details about this code with the help of Figure 11.

Let's start in the diagram with the User. The user has two different actions in the browser: enter a url or enter a search query. When the application first has to be accessed, a URL has to be entered. That action is received by the Django web server which is named Tesla for

⁸www.djangoproject.com

Figure 11: The internal structure of the Django setup



this application. A server in Django can have several applications, but our system has only one application called **tessearch**. This is not represented in the diagram, because this is not important for the internal structure, because our system has only one application. Our server is named Tesla, and when this server is started, the **settings.py** file is read and all the settings are loaded. Another file in the main Tesla folder is called **urls.py**. This file contains all the information for the server what to do with incoming urls. When the server is accessed for the first time, the server redirects the browser of the user to the **base.html** found in the **templates** folder in the aforementioned **tessearch** application.

The **base.html** is a composite page of different HTML pages. The **base.html** load all the important tags and other files for Django. In the diagram, this can be seen by noticing the relationship with the **Static** class it uses; all the css files and images can be found here. The **search.html** is the first page found when going to the server. This page is basically just the searchbar with no results. The user can then type in the search query in this page, which will then load the **searchwithresults.html** file. This page loads in up to four different result pages, depending on what documents are found with the search query.

These results are requested by the html pages from the database, the **Model**, as can be seen in the diagram. Following this, the search results are returned in a custom **TeslaSearchForm**, which is a subclass of the standard Haystack **SearchForm**. Before finally returning these search results, the form is used by the **TextHighlighter**, which is a subclass of the standard Haystack **HighLighter** class, which highlights search results. When these search results are shown in a list of results in **searchwithresults.html**, they first have to be retrieved from the local database to call on custom methods and attributes defined in **models.py**. When these objects are requested,

an object is returned which can be used by the Django code in the custom html pages.

When a result is clicked, the application shows a local result. To do this, a html page displays a hyperlink reference to a local URL. This reference, as can be seen in the diagram, is interpreted by **urls.py** and redirects to a new view in **views.py**. This view is made with the object from the database and is displayed in a custom style dictated by **style.css** in the **Static** folder.

The internal structure of Django, thus the View component of the MVC, is pretty complicated to understand when starting. Especially in the beginning, we had a lot of trouble getting basic things to work. But, as we began understanding more and more of the internal structure, even more advanced functions became easier to implement.

TODO

4.3.2 UI

One of the key requirements was to keep the system as easy as possible. Anyone who could boot a computer should be able to use the system. In this section the structure of the HTML pages will explained.

Overall look-and-feel

The overall look-and-feel should be clean and minimalistic with no distraction. Chosen was the Bootstrap 3 CSS library to design the look-and-feel. It has been used for the overall web application.

Template structure

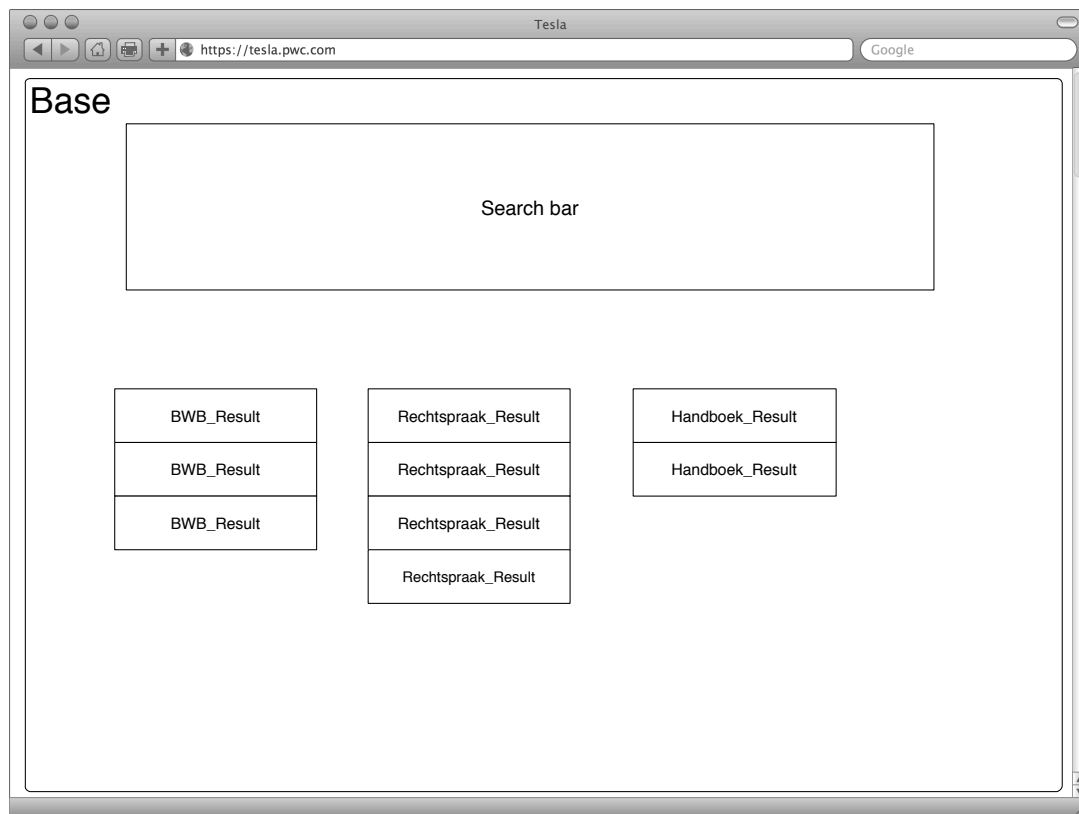
All the webpages have their own template which Django renders to a valid HTML page. All pages inherit from a Base.html page, in which other template parts are included. The templates contain their own JavaScript that has to interact with different parts in the Template.

There are two main templates that can be enriched with a few different components:

- Search-template.
Renders the Home-page, only inherits the searchbar
- SearchWithResults-template Renders the Results Page. Contains the searchbar and a result template for every source.

By keeping the same inheriting-methodology of Software Engineering within the Django templates, a structured and organised design could be guaranteed for both development and usability. Below is a visual representation of the template structure of the Results Page.

Figure 12: Download Modules



JavaScript manipulation of Results

The JavaScript in the templates create a dynamic rendering of the result page. The User should be able to hide or show results on the fly. The JavaScript in the HTML page makes sure that the categories chosen in the Search Page are shown in the Result Page. JavaScript hides or shows the results Django creates accordingly.

Chosen for this approach was the requirement that the system should be fast. Users do not want to fire a new search query when hiding irrelevant results based on the outcome of the search query. This would significantly slow down the system and harm the usability.

4.4 Back-end

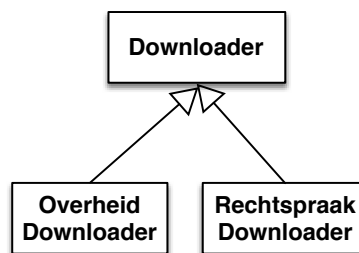
The back-end (Control) in the system takes care of the whole system. It is in control of the Django webserver as well as the Solr search engine and makes sure that it is always live. It drives the Download and Convert module and makes sure that all of them are called in the right order at the right time. The difficult parts in this part of the system were the autonomy, the exception handling and the file system handling. The subsections below will go into further detail.

4.4.1 Downloader Modules

The Downloader module has the task to keep the system up-to-date. It checks for new and updated documents from the four sources, and makes sure that the database of the system is in sync of the changes made at the sources. The Control Module lets the Downloader check the new sources around midnight, to ensure that the usage of both the system and the servers of the sources are not being hammered by the demand of the systems request in combination with other users of the sources.

Every source has its own Downloader Module, that inherits from the Downloader class. Below is a class diagram of the Downloader class structure.

Figure 13: Download Modules



OverheidDownloader

The OverheidDownloader Module is a module that downloads the Basis Wetten from overheid.nl. It does so by downloading an XML file that contains all the BWBID's with their relevant information. OverheidDownloader Module then goes ahead and creates download links from these BWBID's and a base link, after which it downloads all the BWB XML files using python's urllib module. The only way to download the BWB XML files from overheid.nl is by a deeplink supplied by the website. OverheidDownloader constructs this deeplink and downloads them into the Control's `data/overheid/xml` directory for conversion.

In addition to that, OverheidDownloader also downloads the HTML version of the BWB wetten. These files are zipped and thus extracted by the Downloader and stored in their corresponding `data/overheid/xml` directory for conversion.

RechtspraakDownloader

Since rechtspraak.nl also distributes its documents only via a deeplink, the RechtspraakDownloader Module works the same. It downloads an XML file of all the information on the new and updated 'uitspraken' via an XML file. [Rechtspraak.nl](http://rechtspraak.nl) only permits a download of 500 XML files at a time, so a deeplink this has to be kept in mind when downloading more than 500 files. It downloads and saves the downloaded XML 'uitspraken' to `data/rechtspraak/xml`

Belastingdienst & Loonheffingen.nl

Due to the complex format and unavailability of OpenData API's of belastingdienst.nl and loonheffingen.nl, it was not possible to develop a web crawler that automatically downloads and

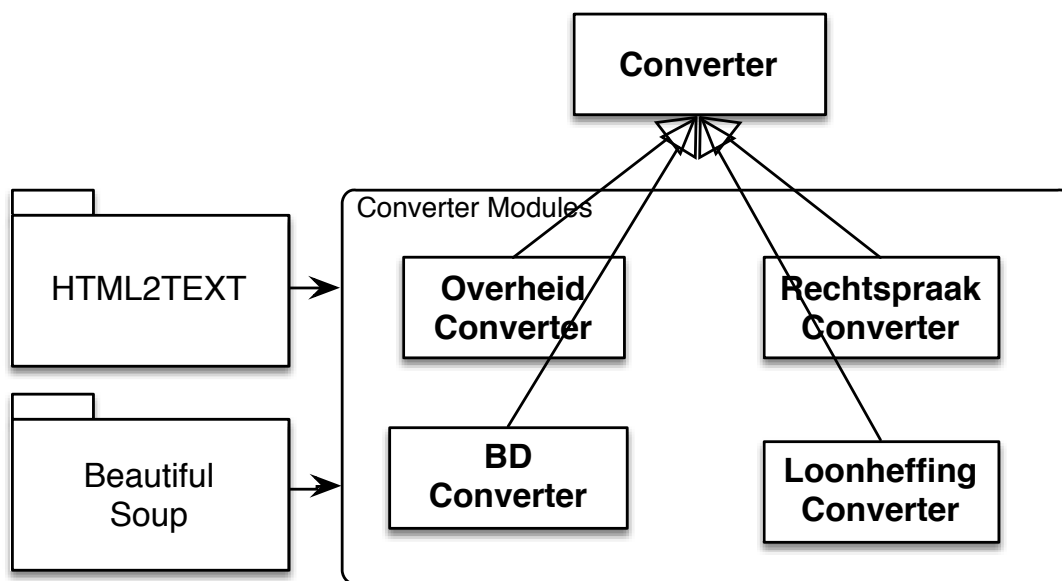
filters the desired content from these site. This may not be such a big issue, since the content on belastingdienst.nl, the ‘Handboek Loonheffingen’, does only change once a years. Content may be downloaded manually and be put in the proper directory, for conversion.

Loonheffingen.nl also updates it documents once a week. This may also be updated manually.

4.4.2 Converter Modules

The Converter Modeles make sure that the different formatted XML and HTML files are prepared for importation into the Model that the Solr database uses, described in section 4.2. This conversion process is more complex than the Download process, and makes use of two 3rd party, open source library, HTML2TEXT & BeautifulSoup.

Figure 14: Converter Modules



Solr XML format

Every single document of the sources had to be converted to the Solr scheme described in section 4.2. HTML files need to be filtered to plain text, this is what the HTML2TEXT library does. The plain text is then content of the Solr XML scheme file.

Other relevant data to fill the scheme is extracted using Python's XML ElementTree module. This conversion process is the same for both *overheid.nl* and *rechtspraak.nl*.

Django HTML

The BWB HTML files have to be modified in order to be found and used by the Django Framework. The location of our own CSS files has to be changed and some Django identifiers have to be added. BeautifulSoup allows the Converter Modules to modify these HTML files.

Belastingdienst & Loonheffingen.nl

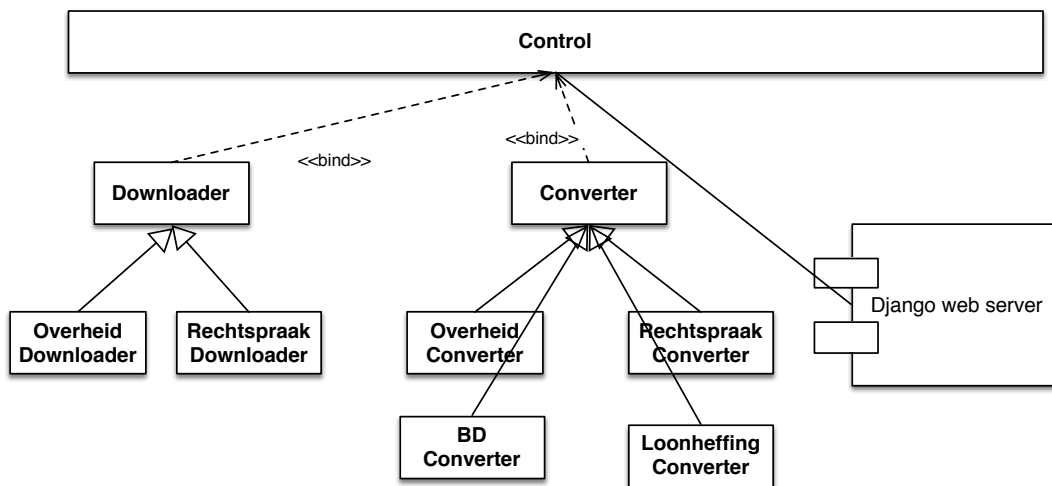
The other HTML formats of the other files are either too complex or undistracting enough and thus not needed to be changed. The HTML files of both are stored locally and can be redirected to from within Django web server.

4.4.3 Control

The Control Module is the module that drives the complete system. That is, the web server, the Solr index engine, the 'uitspraak' relations building, and the Downloader and Converter Modules. The Control Module makes sure that the all different Downloader Modules are initialised, and executed at the right time. After which it fires the Converter Modules to convert the files to the proper formats. Control then fires the Django import, relation building, and index rebuilder scripts to import everything into the Django database.

In the meanwhile it controls the web server in a thread, as shown below in the UML diagram.

Figure 15: Results Page



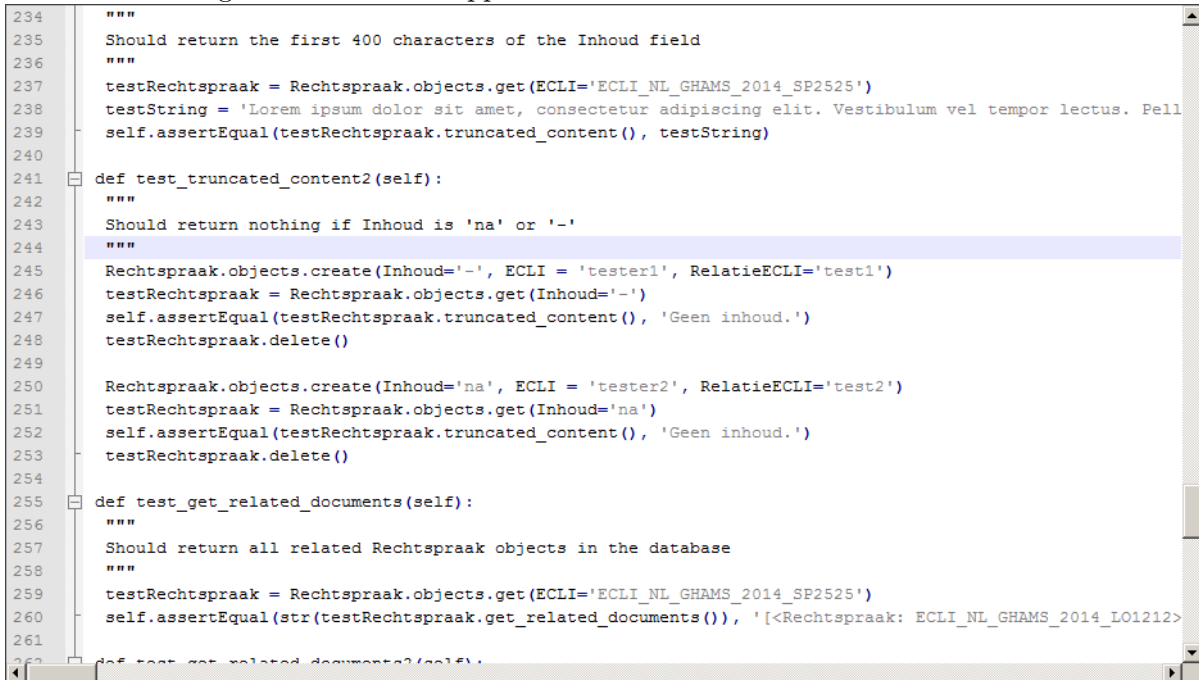
5 Testing

An important part of the system is testing: checking to make sure that the system fits the requirements. There are many ways to do this, but we focused mainly on two different types of testing: automated testing and acceptance testing. These approaches will be fully explained in the next sections.

5.1 Automated Testing

For this system, a number of automated tests were written. These automated tests can also be called unit tests. These tests test a single part of the application, a unit, to ensure that the code meets the design and acts like intended. To this end, the Django **unittest** module was used. In this module, a collection of test cases can be made and executed very easily. In unit testing, Web applications are hard to test because they are made of several layers of logic. However, the test-execution framework of Django can simulate requests, insert test data, mock a database and inspect output from a document to verify code. An example code snippet can be seen in Figure 16.

Figure 16: A code snippet from the test suite of several testcases.



```
234 """
235 Should return the first 400 characters of the Inhoud field
236 """
237 testRechtspraak = Rechtspraak.objects.get(ECLI='ECLI_NL_GHAMS_2014_SP2525')
238 testString = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum vel tempor lectus. Pell
239 self.assertEqual(testRechtspraak.truncated_content(), testString)
240
241 def test_truncated_content2(self):
242     """
243     Should return nothing if Inhoud is 'na' or '-'
244     """
245     Rechtspraak.objects.create(Inhoud='-', ECLI = 'tester1', RelatieECLI='test1')
246     testRechtspraak = Rechtspraak.objects.get(Inhoud='-')
247     self.assertEqual(testRechtspraak.truncated_content(), 'Geen inhoud.')
248     testRechtspraak.delete()
249
250     Rechtspraak.objects.create(Inhoud='na', ECLI = 'tester2', RelatieECLI='test2')
251     testRechtspraak = Rechtspraak.objects.get(Inhoud='na')
252     self.assertEqual(testRechtspraak.truncated_content(), 'Geen inhoud.')
253     testRechtspraak.delete()
254
255 def test_get_related_documents(self):
256     """
257     Should return all related Rechtspraak objects in the database
258     """
259     testRechtspraak = Rechtspraak.objects.get(ECLI='ECLI_NL_GHAMS_2014_SP2525')
260     self.assertEqual(str(testRechtspraak.get_related_documents()), ' [<Rechtspraak: ECLI_NL_GHAMS_2014_LO1212>
261
262 def test_get_related_documents2(self):
```

The framework was used mostly to check if the local SQLite database behaved correctly, and thus that the Django models were implemented correctly. These tests were the most important because if the models could return the right document, then the HTML templates could also return the correct documents by calling an object instance. And when the database was correctly configured, Solr also had the right documents in its database, as the Solr database basically mirrored the Django database. Thus, exhaustive testcases were written to confirm that the objects retrieved from the SQLite database fit the requirements.

This is especially true for the **Rechtspraak** model in the database. This model has some extra methods that have to be thoroughly tested, mostly because the client has expressed primary interest in these methods. The most important method is the **get_related_documents** method. This method collects all related documents of a document and returns them in a list. This includes

not only documents that document references in its field **RelatieECLI**, but also all documents that have a reference to this document. This has not yet been implemented in an earlier system and thus has to be exhaustively tested.

To do this, the Django **unittest** module was used to mock the SQLite database, creating mocked entries of various different relation hierarchies. Also, different 'empty' database entries were also made to test if appropriate warnings and responses were given and the system did not fail. An example of a complicated relation hierarchy can be seen in Figure 17.

Figure 17: A code snippet from the test suite showing a mocked complicated relation hierarchy of Rechtspraak documents.

```
#####  
# Advanced testcases for recursion #  
#####  
Rechtspraak.objects.create(ECLI='C0', RelatieECLI='A0;C1;C2', Sorteerdatum='2014-03-24')  
Rechtspraak.objects.create(ECLI='C1', RelatieECLI='C0', Sorteerdatum='2014-03-23')  
Rechtspraak.objects.create(ECLI='C2', RelatieECLI='C0', Sorteerdatum='2014-03-22')  
Rechtspraak.objects.create(ECLI='A0', RelatieECLI='A1;B0;C0;A2', Sorteerdatum='2014-03-30')  
  
Rechtspraak.objects.create(ECLI='A1', RelatieECLI='A0', Sorteerdatum='2014-03-29')  
Rechtspraak.objects.create(ECLI='A2', RelatieECLI='A0', Sorteerdatum='2014-03-28')  
  
Rechtspraak.objects.create(ECLI='B0', RelatieECLI='A0;B1;B2', Sorteerdatum='2014-03-27')  
  
Rechtspraak.objects.create(ECLI='B1', RelatieECLI='B0', Sorteerdatum='2014-03-26')  
  
Rechtspraak.objects.create(ECLI='B2', RelatieECLI='B0', Sorteerdatum='2014-03-25')
```

An important note is that the searching through the database for related documents is not done in the model, but in Solr. The related documents are gathered by a one-time pass algorithm running over the entire database, ensuring that every related document is found. This was done because if the algorithm was done in real-time, when a search query was processed, the results would return very slow. This way, all the data is preprocessed and is returned much faster.

5.2 Acceptance Testing

One of the more important types of testing is the acceptance testing. Acceptance testing is conducted to determine if the requirements of the client are met. These acceptance tests with the client were held almost every week, during each development cycle. These were more informal acceptance tests, often consisting of small demo's for different people in the Human Resource Support department. These weekly demo's were very important for finalising and confirming requirements, and even more important to decide on what functions the project should focus. As another plus, these acceptance tests also helped to bring the client up to date with how the system progressed.

Aside from these weekly informal acceptance tests, two formal acceptance tests were planned during the development process. The first acceptance test was planned after four weeks, when

the basic system was up and running. The main reason for this acceptance test was to check what features the client wanted to see exactly in the final product. The second acceptance was planned at the end of development, just before the final presentation. The main reason for the final acceptance test was to check if the final product fits the requirements. These two acceptance tests, the half-way acceptance test and the final acceptance tests, will be explained further in the next sections.

5.2.1 Half-way acceptance test

For the half-way acceptance test, we presented a demo for the client in a meeting. This meeting was attended by our main client and a colleague, both interested in using the tool.

At this stage, we particularly focused on acceptance testing the basic functionality of the system. The users were expected to use their own creativity to come up with interesting search queries, as they are experts on the subject. Below are the questions listed that were individually asked to the two users:

- Try to find your way around, do you understand
- Is content that you would expect find in the system, in there?
- Can you find the documents that you would look for using the search query?
- Did the system behave abnormally, not as expected, during the tests?
- Did the system serve irrelevant or odd results during the test cases?

The users were positive during these basic acceptance test. Sometimes the system produced odd results, but this was caused by an unindexed date field with made the system return old and unlikely to be relevant documents.

5.2.2 Final acceptance test

In the last fase of the project, the final acceptance tests were preformed. We added a third user to improve usability. This user has never seen the system before and is also not familiar with the content of the system. We will ask the user the same questions and compare the results to the users that are familiar with the content of the search tool.

The same questions of the initial acceptance test were asked to ensure that the preformed development was in line with earlier results. After positive feedback, the following tests where preformed. Particularly tests that can not be verified by the team for developers because of the nature of the content serviced by the system.

- Is the chaining of the ‘uitspraken’ correctly represented?
- Is the most complexed case you know chained correctly?
- Are the ‘Beleid’ documents displayed correctly (since this sources was added in a later stage).
- Do the logical operators work correctly?
- Find the document in which ‘Auto van de zaak’ is the most relevant query and find the corresponding ‘Hoge Raad uitspraak’.

At the writing of this document, this last acceptance test had yet to take place. Results will be disclosed in the final presentation.

5.3 SIG Feedback

After seven weeks of development, the code of the project was submitted to the Software Improvement Group, SIG ⁹. SIG is a company that identifies the complexity of software and advises clients where to optimise code to lower costs and lower the chance of errors. The code submitted to SIG was code from the Controller component of the system: the format converter code and downloader code. The reason we only submitted the Controller code was that the Model consisted only of databases, which can not be checked for complexity. The view consisted primarily of Django Framework code, and as the code of this View consisted of multiple individual scripts this would be hard for SIG to measure complexity too. So, the only code submitted was the code in the Control component.

This code rated three stars out of five on the maintainability scale of SIG. In this section, a summary is made from the comments. The reason the code did not get a higher score was that the code scored badly on the subjects Duplication en Unit Complexity. The full feedback can be seen in Appendix B 8 .

5.3.1 Feedback on Duplication

For duplication, the code is checked for redundancy, code that is present in the system multiple times. When a system has much redundancy, the maintainability decreases. Because, if code has to be changed, then multiple sections have to change at the same time in which errors can occur. This happened in the code of **format_converter.py**. An example of this redundancy is given in Figure 18.

In figure 19, the code seen is used to convert documents into a format that can be index in the SQLite databases. Although two different types of documents are converted, the code looks very much alike. This code thus has a high rate of redundancy, and has lower maintainability.

5.3.2 Feedback on Unit Complexity

For unit complexity, the code is checked for the percentage of code that has a complexity that is above average. Complexity is decreased when large methods are split in smaller methods, making it easier to read and understand the code. In the method **transform_to_solr_scheme()** the complexity was very high. This can be seen in figure 19.

This method essentially boils down to a large switch-case, which is hard to read. In addition, for reading out the relation of a document a complex if-else statement is embedded in the switch-case, raising the complexity further.

5.3.3 Improvements based on feedback

Essentially, there are two pieces of code where the complexity has to be lowered. To lower the redundancy, a more object-oriented approach should be done. This way, copying code is avoided, because a specific converter can then be a subclass of a converter interface. This will also raise the maintainability. The second problem, the unit complexity, has an easy solution. Instead of write large pieces of code, a separate method can be created that accepts appropriate arguments for every type of converter.

These solutions are not implemented yet by the time of writing this paper; the results will be disclosed during the final presentation.

⁹<http://www.sig.eu/en/>

Figure 18: A code snippet from format_convert.py showing duplication (redundancy) in the code.

```

545 """
546 This class converts HTML files from belastingdienst to Solr XML. HTML files should
547 """
548 class BDConverter():
549
550     def __init__(self, destination_path):
551         print 'BDConverter init: ' + str(destination_path)
552         try :
553             os.makedirs(os.path.join(destination_path, 'BD/xml/'))
554             os.makedirs(os.path.join(destination_path, 'BD/html/'))
555         except OSError as err:
556             if err.errno!=17:
557                 raise
558             else:
559                 print "[INFO] Dir BD/html/ already exists."
560         self.destination = os.path.abspath(os.path.join(destination_path, 'BD/'))
561         self.html_dir = os.path.join(self.destination, 'html/')
562         self.xml_dir = os.path.join(self.destination, 'xml/')
563
343 """
344 This class converts ECLI XML to Solr XML
345 """
346 class RechtspraakConverter():
347
348     def __init__(self, destination_path):
349         self.destination = destination_path
350         print 'Rechtspraak init: ' + str(self.destination)
351         try :
352             os.makedirs(os.path.join(destination_path, 'rechtspraak/xml/'))
353         except OSError as err:
354             if err.errno!=17:
355                 raise
356             else:
357                 print "[INFO] Dir rechtspraak/xml/ already exists."
358         self.destination = os.path.abspath(os.path.join(destination_path, 'rechtspraak/'))
359         self.xml_dir = os.path.join(self.destination, 'xml/')
360

```

Figure 19: A code snippet from format_convert.py showing duplication (redundancy) in the code.

```

410 #issuedSemaphore is used to retrieve only the first issue date, because that is the most recent one
411 if child.tag == '{http://purl.org/dc/terms/}issued' and issuedSemaphore:
412     fields['Uitgavedatum'] = child.text
413     issuedSemaphore = False
414 if child.tag == '{http://purl.org/dc/terms/}relation':
415     #if there is no other Relation ECLI already found
416     if fields['RelatieECLI'] == 'na':
417         #first, split the first relation type info from the contents
418         relatieList = child.text.strip().split(':')
419         fields['RelatieType'] = relatieList[0]
420         relatieList = relatieList[1:]
421
422     #if there is some other relation info at the end
423     if ',' in relatieList[-1]:
424         #slice the info and paste it with the RelatieType
425         fields['RelatieType'] += relatieList[-1].split(',')[1]
426         relatieList[-1] = relatieList[-1].split(',')[0]
427     #finally, extract the relation ECLI
428     fields['RelatieECLI'] = (':' + ''.join(relatieList)).strip()
429     #if another relation ECLI is found, append stuff with ; between them
430     else:
431         relatieList = child.text.strip().split(':')
432         #an other relation has been found, so be sure to separate it with ;
433         fields['RelatieType'] += ';'
434         fields['RelatieType'] += relatieList[0]
435         relatieList = relatieList[1:]
436
437     #if there is some other relation info at the end
438     if ',' in relatieList[-1]:
439         #slice the info and paste it with the RelatieType
440         #no ; has to be added, it's still part of the new Relation Type
441         fields['RelatieType'] += relatieList[-1].split(',')[1]
442         relatieList[-1] = relatieList[-1].split(',')[0]
443     #append new relation ECLI with ;
444     fields['RelatieECLI'] += ';'
445     fields['RelatieECLI'] += (':' + ''.join(relatieList)).strip()
446
447 if child.tag == '{http://purl.org/dc/terms/}publisher':
448     fields['Uitgever'] = child.text
449 if child.tag == '{http://purl.org/dc/terms/}date':
450     fields['Uitspraakdatum'] = child.text
451 if child.tag == '{http://purl.org/dc/terms/}modified' :
452     fields['DatumLaatsteWijziging'] = child.text
453 if child.tag == '{http://purl.org/dc/terms/}language':
454     fields['Taal'] = child.text

```

6 Future Work & Improvements

Even though the most of the requirements have been met after the scheduled time of the project, there is still a lot of work has to be done to bring this Proof of Concept to an actual search engine in production. This section will highlight a few points of improvement and future work that should be done to complete Tesla and bring it to a Google-competitive level of service.

6.1 Django and Hackstack

The Django SQLite database was, in hindsight, not fit for handling over 50.000 documents. Even though the Solr engine is lightning fast, we pushed Django's SQLite database to its limits and beyond. The relation building of the 'uitspraken' was a very intensive task using the tools and API from Django. A more mature framework should be used to interfere with the database. Not being able to fire SQL statements to the database is absolutely not desirable; this was handled in the project by Haystack. A more mature DBMS should be used to process the queries.

6.2 Optimalisations

Especially in the script that builds relations can be improved a lot of terms of speed and performance. There was not enough time to fully optimise this algorithm.

Also the result generation should be faster. We have not yet found out what the bottleneck is here, but very likely, this is the result of inefficient SQL statement generation by Haystack.

6.3 Failure proof

The back-end system could is arguably the heart of the system. A system that is supposed be live 24/7 while updating itself with new documents from the specified sources.

During the project a certain amount of stability of the back-end has been reached, but this should be improved. For a 24/7 uptime, more exception handling should be developed and more exceptional cases of failures of connected services should be tested and handled.

6.4 Web crawler

Currently Tesla indexes four sources, but can only automatically download two of them, overheid.nl and rechspraak.nl, since they provide some sort of OpenData API. Loonheffingen.nl and belastingdienst.nl do not provide this feature publicly. Due to time limitations and platform compatibility, it was not possible to implement a web crawler that automatically crawles and downloads new content from those sources.

A third party webcrawler like the Python based Mechanize module may be used to implement this feature in the future. If done so, the back-end system can be easily extended with this functionality.

6.5 Sorting options

Currently the the system sorts the results based on relevance. An option is available to sort on date, but a combination of both would be better.

In a future release of the system should feature the ability for the user to sort results based on date. This would allow users to sort results primarily on date and thus get for example the latest 'uitspraken' at the top of the results.

Conclusion

In this Bachelor Project a custom Search Engine called Tesla has been developed. It is a search engine that indexes and serves four different sources that are regularly used by PwC professionals. In particular, the professionals of the wage taxes practice of PwC NL.

The system was developed by following a scientific and agile approach using Scrum, dividing the process in several phases. First, meetings were held with the domain experts and a specification and requirements were made based on the assignment. Next, a literature survey was performed to determine what techniques to use during the project, and an initial design of the system was made. Then, the system was implemented incrementally using Scrum. Finally, the system was tested to ensure correct functionality and to ensure the set requirements were met.

Initially a few requirements were set by the client. We enriched that set of requirements with our own set of requirements. The most important requirements and their solution are described in this section.

Speed

‘The system had to be fast, Google like search results.’

This was a hard one. Keeping a Python system based on Haystack with a Java search engine with over 50.000 interconnected documents fast was a challenge. With a maximum waiting time of 10 seconds, we found that this requirement was met. Even though, as described in the Future Work section, another framework to manage the search results should have been used.

Usability

‘The system should be easy to use for someone that has minimum computer knowledge’.

By going for a minimalistic design, and minimising options a user has to make, we felt that we met the requirement. Acceptance tests confirmed this.

Sources

‘The system should serve content from 4 predefined sources and update automatically.’

Currently the system is able to automatically update from two sources, overheid.nl and recht-spraak.nl, and able to convert and index all four of them. Since two of them are not ‘open’ enough and time was limiting, content of these two will have to be added manually to the system.

Our own specified requirements have all been met and thereby successfully completed system. We enjoyed working with PwC and want to thank them for the opportunity.

7 Appendix A: The schema for the Solr database

Schema.xml.

```
1  <?xml version="1.0" ?>
   <!--
3   Licensed to the Apache Software Foundation (ASF) under one or more
   contributor license agreements. See the NOTICE file distributed with
5   this work for additional information regarding copyright ownership.
   The ASF licenses this file to You under the Apache License, Version 2.0
7   (the "License"); you may not use this file except in compliance with
   the License. You may obtain a copy of the License at
9
       http://www.apache.org/licenses/LICENSE-2.0
11
   Unless required by applicable law or agreed to in writing, software
13   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15   See the License for the specific language governing permissions and
   limitations under the License.
17   -->

19  <schema name="default" version="1.4">
    <types>
21      <fieldtype name="string" class="solr.StrField" sortMissingLast="true" o
        <fieldType name="boolean" class="solr.BoolField" sortMissingLast="true" o
23      <fieldtype name="binary" class="solr.BinaryField"/>

25      <!-- Numeric field types that manipulate the value into
           a string value that isn't human-readable in its internal form,
27           but with a lexicographic ordering the same as the numeric ordering,
           so that range queries work correctly. -->
29      <fieldType name="int" class="solr.TrieIntField" precisionStep="0" omitNo
        <fieldType name="float" class="solr.TrieFloatField" precisionStep="0" om
31      <fieldType name="long" class="solr.TrieLongField" precisionStep="0" omit
        <fieldType name="double" class="solr.TrieDoubleField" precisionStep="0" o
33
        <fieldType name="tint" class="solr.TrieIntField" precisionStep="8" omitN
35      <fieldType name="tfloat" class="solr.TrieFloatField" precisionStep="8" o
        <fieldType name="tlong" class="solr.TrieLongField" precisionStep="8" omi
37      <fieldType name="tdouble" class="solr.TrieDoubleField" precisionStep="8"

39      <fieldType name="date" class="solr.DateField" omitNorms="true" positionI
        <!-- A Trie based date field for faster date range queries and date face
41      <fieldType name="tdate" class="solr.TrieDateField" omitNorms="true" prec

43      <fieldType name="point" class="solr.PointType" dimension="2" subFieldSuff
        <fieldType name="location" class="solr.LatLonType" subFieldSuffix="_coo
45      <fieldtype name="geohash" class="solr.GeoHashField"/>

47      <fieldType name="text\_general" class="solr.TextField" positionIncrement
```

```

49     <tokenizer class="solr.StandardTokenizerFactory"/>
51     <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords" />
53     <!-- in this example, we will only use synonyms at query time
55     <filter class="solr.SynonymFilterFactory" synonyms="index/_synonyms.txt" />
57     <filter class="solr.LowerCaseFilterFactory"/>
59 </analyzer>
61 <analyzer type="query">
63     <tokenizer class="solr.StandardTokenizerFactory"/>
65     <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords" />
67     <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt" ignoreCase="true" />
69     <filter class="solr.LowerCaseFilterFactory"/>
71 </analyzer>
73 </fieldType>

<fieldType name="text/_en" class="solr.TextField" positionIncrementGap="100">
75     <analyzer type="index">
77         <tokenizer class="solr.StandardTokenizerFactory"/>
79         <filter class="solr.StopFilterFactory"
81             ignoreCase="true"
83             words="stopwords/_en.txt"
85             enablePositionIncrements="true"
87             />
89         <filter class="solr.LowerCaseFilterFactory"/>
91         <filter class="solr.EnglishPossessiveFilterFactory"/>
93         <filter class="solr.KeywordMarkerFilterFactory" protected="protwords" />
95         <!-- Optionally you may want to use this less aggressive stemmer instead
97         <filter class="solr.EnglishMinimalStemFilterFactory"/>
99         <!-->
101        <filter class="solr.PorterStemFilterFactory"/>
102    </analyzer>
103    <analyzer type="query">
105        <tokenizer class="solr.StandardTokenizerFactory"/>
107        <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt" ignoreCase="true" />
109        <filter class="solr.StopFilterFactory"
111            ignoreCase="true"
113            words="stopwords/_en.txt"
115            enablePositionIncrements="true"
117            />
119        <filter class="solr.LowerCaseFilterFactory"/>
121        <filter class="solr.EnglishPossessiveFilterFactory"/>
123        <filter class="solr.KeywordMarkerFilterFactory" protected="protwords" />
125        <!-- Optionally you may want to use this less aggressive stemmer instead
127        <filter class="solr.EnglishMinimalStemFilterFactory"/>
129        <!-->
131        <filter class="solr.PorterStemFilterFactory"/>
132    </analyzer>
133 </fieldType>

<fieldType name="text/_ws" class="solr.TextField" positionIncrementGap="100">

```

```

99         <tokenizer class="solr.WhitespaceTokenizerFactory"/>
101     </tokenizer>
102 </fieldType>
103
104 <fieldType name="ngram" class="solr.TextField" >
105     <tokenizer type="index">
106         <tokenizer class="solr.KeywordTokenizerFactory"/>
107         <filter class="solr.LowerCaseFilterFactory"/>
108         <filter class="solr.NGramFilterFactory" minGramSize="3" maxGramSize=
109     </filter>
110     </tokenizer>
111     <tokenizer type="query">
112         <tokenizer class="solr.KeywordTokenizerFactory"/>
113         <filter class="solr.LowerCaseFilterFactory"/>
114     </filter>
115     </tokenizer>
116 </fieldType>
117
118 <fieldType name="edge\_ngram" class="solr.TextField" positionIncrementGap="100">
119     <tokenizer type="index">
120         <tokenizer class="solr.WhitespaceTokenizerFactory" />
121         <filter class="solr.LowerCaseFilterFactory" />
122         <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
123         <filter class="solr.EdgeNGramFilterFactory" minGramSize="2" maxGramSize="100" />
124     </filter>
125     </tokenizer>
126     <tokenizer type="query">
127         <tokenizer class="solr.WhitespaceTokenizerFactory" />
128         <filter class="solr.LowerCaseFilterFactory" />
129         <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
130         <filter class="solr.EdgeNGramFilterFactory" minGramSize="2" maxGramSize="100" />
131     </filter>
132     </tokenizer>
133 </fieldType>
134 </types>
135
136 <fields>
137     <!-- general -->
138     <field name="id" type="string" indexed="true" stored="true" multiValued="false"/>
139     <field name="django\_ct" type="string" indexed="true" stored="true" multiValued="false"/>
140     <field name="django\_id" type="string" indexed="true" stored="true" multiValued="false"/>
141
142     <dynamicField name="*\_i" type="int" indexed="true" stored="true"/>
143     <dynamicField name="*\_s" type="string" indexed="true" stored="true"/>
144     <dynamicField name="*\_l" type="long" indexed="true" stored="true"/>
145     <dynamicField name="*\_t" type="text\_en" indexed="true"
146     stored="true"/>
147     <dynamicField name="*\_b" type="boolean" indexed="true" stored="true"/>
148     <dynamicField name="*\_f" type="float" indexed="true" stored="true"/>
149     <dynamicField name="*\_d" type="double" indexed="true" stored="true"/>
150     <dynamicField name="*\_dt" type="date" indexed="true" stored="true"/>
151     <dynamicField name="*\_p" type="location" indexed="true" stored="true"/>
152     <dynamicField name="*\_coordinate" type="tdouble" indexed="true"
153     stored="false"/>

```


149
151
153
155
157
159
161
163
165
167
169
171
173
175
177
179
181
183
185
187
189
191
193
195
197

```
<field name="Rechtsgebied" type="text\_en" indexed="true" stored="true" m
<field name="Inhoudsindicatie" type="text\_en" indexed="true" stored="tru
<field name="Inhoud" type="text\_en" indexed="true" stored="true" multiVal
<field name="DatumLaatsteWijziging" type="text\_en" indexed="true" stored
<field name="Titel" type="text\_en" indexed="true" stored="true" multiVal
<field name="text" type="edge\_ngram" indexed="true" stored="true" multi
<field name="Taal" type="text\_en" indexed="true" stored="true" multiVal
<field name="Uitgever" type="text\_en" indexed="true" stored="true" mult
<field name="Uitspraakdatum" type="text\_en" indexed="true" stored="true
<field name="Zaaknummer" type="text\_en" indexed="true" stored="true" mu
<field name="Vervangt" type="text\_en" indexed="true" stored="true" mult
<field name="ECLI" type="text\_en" indexed="true" stored="true" multiVal
<field name="Bron" type="text\_en" indexed="true" stored="true" multiVal
<field name="Type" type="text\_en" indexed="true" stored="true" multiVal
<field name="Procedure" type="text\_en" indexed="true" stored="true" mul
<field name="Status" type="text\_en" indexed="true" stored="true" multiVal
<field name="CiteerTitel" type="text\_en" indexed="true" stored="true" m
<field name="VervalDatum" type="text\_en" indexed="true" stored="true" m
<field name="InwerkingsTredingsDatum" type="text\_en" indexed="true" stor
<field name="OfficiëleTitel" type="text\_en" indexed="true" stored="true
<field name="RegelingSoort" type="text\_en" indexed="true" stored="true"
<field name="Afkorting" type="text\_en" indexed="true" stored="true" mul
<field name="RegelingInfo" type="text\_en" indexed="true" stored="true" m
<field name="Jaar" type="text\_en" indexed="true" stored="true" multiVal
<field name="HBLH" type="text\_en" indexed="true" stored="true" multiVal
```

199
201
203
205
207
209
211
213
215
217
219
221
223
225
227
229
231
233
235
237
239
241
243
245
247
249

```
<field name="Uitgavedatum" type="text\_en" indexed="true" stored="true" m
<field name="Maker" type="text\_en" indexed="true" stored="true" multiVal
<field name="RelatieType" type="text\_en" indexed="true" stored="true" m
<field name="RelatieECLI" type="text\_en" indexed="true" stored="true" m
<field name="BesluitID" type="text\_en" indexed="true" stored="true" mul
<field name="Gewijzigd" type="text\_en" indexed="true" stored="true" mul
<field name="SorteerDatum" type="date" indexed="true" stored="true" mult
</fields>

<!-- field to use to determine and enforce document uniqueness. -->
<uniqueKey>id</uniqueKey>

<!-- field for the QueryParser to use when an explicit fieldname is absent
<defaultSearchField>text</defaultSearchField>

<!-- for letting solr search in every field, not just text-->
<!-- for Wet model -->
<copyField source="Status" dest="text"/>
<copyField source="CiteerTitel" dest="text"/>
<copyField source="DatumLaatsteWijziging" dest="text"/>

<copyField source="Titel" dest="text"/>
<copyField source="VervalDatum" dest="text"/>
<copyField source="InwerkingsTredingsDatum" dest="text"/>

<copyField source="Bron" dest="text"/>
<copyField source="OfficiëleTitel" dest="text"/>
<copyField source="RegelingSoort" dest="text"/>

<copyField source="Afkorting" dest="text"/>
<copyField source="RegelingInfo" dest="text"/>
<copyField source="Inhoud" dest="text"/>

<!-- for rechtspraak model -->
<copyField source="ECLI" dest="text"/>
<copyField source="Inhoudsindicatie" dest="text"/>
<copyField source="DatumLaatsteWijziging" dest="text"/>
<copyField source="Uitgever" dest="text"/>

<copyField source="Taal" dest="text"/>
<copyField source="Vervangt" dest="text"/>
<copyField source="Uitspraakdatum" dest="text"/>

<copyField source="Zaaknummer" dest="text"/>
```

```

251     <copyField source="Type" dest="text"/>
252     <copyField source="Procedure" dest="text"/>
253
254     <copyField source="Rechtsgebied" dest="text"/>
255     <copyField source="Bron" dest="text"/>
256     <copyField source="Titel" dest="text"/>
257     <copyField source="Inhoud" dest="text"/>
258
259     <copyField source="Uitgavedatum" dest="text"/>
260     <copyField source="Maker" dest="text"/>
261
262     <copyField source="RelatieType" dest="text"/>
263     <copyField source="RelatieECLI" dest="text"/>
264
265     <!-- for handboekpagina model -->
266     <copyField source="HBLH" dest="text"/>
267     <copyField source="Titel" dest="text"/>
268     <copyField source="Bron" dest="text"/>
269     <copyField source="Jaar" dest="text"/>
270     <copyField source="Inhoud" dest="text"/>
271
272     <!-- for besluit model -->
273     <copyField source="BesluitID" dest="text"/>
274     <copyField source="Titel" dest="text"/>
275     <copyField source="Bron" dest="text"/>
276     <copyField source="Jaar" dest="text"/>
277     <copyField source="Gewijzigd" dest="text"/>
278     <copyField source="Inhoud" dest="text"/>
279
280     <!-- SolrQueryParser configuration: defaultOperator="AND|OR" -->
281     <solrQueryParser defaultOperator="AND"/>
</schema>

```

8 Appendix B: Feedback SIG

Beste Vincent Van Mieghem,

Hierbij ontvang je onze evaluatie van de door jou opgestuurde code. De evaluatie bevat een aantal aanbevelingen die meegenomen kunnen worden in de laatste fase van het project.

Deze evaluatie heeft als doel om studenten bewuster te maken van de onderhoudbaarheid van hun code en dient niet gebruikt te worden voor andere doeleinden.

Mochten er nog vragen of opmerkingen zijn dan hoor ik dat graag.

Met vriendelijke groet, Eric Bouwers

Aanbevelingen

De code van het systeem scoort 3 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code gemiddeld onderhoudbaar is. Een hogere score is niet behaald door lagere scores voor Duplication en Unit Complexity.

Voor Duplication wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren. In jullie `format_converter.py` zijn er stukken code die meerdere keren voorkomen, kijk bijvoorbeeld maar eens naar `BDConverter` en `RechtspraakConverter`. Het is aan te raden om dit soort duplicaten op te sporen en te verwijderen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. In `transform_to_solr_scheme()` gebruiken jullie bijvoorbeeld een hele grote switch op basis van element naam om een datastructuur te vullen. Deze code is erg moeilijk testbaar, en je maakt snel een fout omdat er veel herhaling in zit. Je kunt dit makkelijker opschrijven door het gedrag en de logica te scheiden. Bijvoorbeeld:

```
tagToFieldMap = {  
'http://purl.org/dc/terms/creator' : 'Maker',  
'http://purl.org/dc/terms/subject' : 'Rechtsgebied'  
}
```

```
for tag in tagToFieldMap.keys():  
    if child.tag == tag:  
        fieldName = tagToFieldMap[tag]  
        fields[fieldName] = child.text
```

Als laatste nog de opmerking dat er geen (unit)test-code is gevonden in de code-upload. Het is sterk aan te raden om in ieder geval voor de belangrijkste delen van de functionaliteit automatische tests gedefinieerd te hebben om ervoor te zorgen dat eventuele aanpassingen niet voor ongewenst gedrag zorgen.

9 Appendix C: Code for chaining algorithm

```
2  #a set to keep track of the documents already updated
   skipSet = set()
4
   #counter to keep track of documents failed to update
6   failCounter = 0

8   #get all the rechtspraak documents
   rechtspraakList = Rechtspraak.objects.all()[24642:]
10  total = len(rechtspraakList)
   count = 0
12  #for all rechtspraak documents
   for rechtspraak in rechtspraakList:
14      print '[DEBUG] Processing: ' + str(count) + ' of ' + str(total) + ', ECLI
         count+=1
16      #if the document is not already updated and can not be skipped
         if rechtspraak.ECLI not in skipSet:

18          #array to store list of related documents
20          #start recursive call with empty list
         print '[DEBUG] Start recursive call...'
22          relationList = self.find_related_rechtspraak(rechtspraak, set())
         print '[DEBUG] Got results...'
24          #add these relations to the relation field of the document
         for relation in relationList:
26              #don't add the same ECLI twice
                 if relation.ECLI not in rechtspraak.RelatieECLI:
28                     #if there is not a relation yet, make a new one
                         if rechtspraak.RelatieECLI == 'na':
30                             rechtspraak.RelatieECLI = relation.ECLI
                         #else, append it
32                     else:
                         rechtspraak.RelatieECLI += ';'
34                     rechtspraak.RelatieECLI += str(relation.ECLI)

36          #save the changed rechtspraak
         print '[DEBUG] Saving..'
38          rechtspraak.save()
         print '[DEBUG] Saving done. '

40
   #then, since all the relations have been found for all the related docum
42   #also fill these in for the other documents
   relatedList = rechtspraak.RelatieECLI.split(';')
44   for relatedECLI in relatedList:
       if relatedECLI != 'na':
46       try:
           print '[DEBUG] Getting documents by relatedECLI'
48           editDocument = Rechtspraak.objects.get(ECLI=relatedECLI)
           editDocument.RelatieECLI = rechtspraak.RelatieECLI
```

```

50         print '[DEBUG] Saving edited docs'
           editDocument.save()
52     except:
           failCounter += 1
54         print 'Document failed to update: ', relatedECLI
           continue

56
           #tell the algorithm to skip this file when found
58         print '[DEBUG] Updating skipset'
           skipSet = skipSet | set([relatedECLI])
60         print '[DEBUG] skipset updated'

62     print '\nAll related documents found and processed.\n'
        print 'Amount of documents failed to update: ', failCounter
64
        #####
66     ##          MAIN ALGORITHM          ##
        #####

68
def find_related_rechtspraak(self, document, resultSet):
70
    #first, make a set of all the related documents of this document
72    relationSet = set()
    relationList = document.RelatieECLI.split(';')
74    currently = datetime.now().strftime('%H:%M:%S:%MS')
    print '[DEBUG] [' + currently + '] find_related_rechtspraak: looping thru
76    for releCLI in relationList:
        if releCLI != 'na':
78        try:
            relationSet = relationSet | set([Rechtspraak.objects.get(ECLI=releCLI)])
80        except:
            continue
82    print '[DEBUG] find_related_rechtspraak: add all docs referencing this doc
    #add all the documents referencing this document to the relationSet
84    relationSet = relationSet | set(Rechtspraak.objects.filter(RelatieECLI__con

86    #make sure that the document itself is not present in the relations
    relationSet.discard(document)
88
    #now, check if there are documents left that have not yet been added to tl
90    relationSet = relationSet - resultSet

92    #if the document is already present, just return the resultset
    if not relationSet:
94
        resultSet = resultSet | set([document])
96        return resultSet

98    #add this document, for checking if other relations don't reference this d
    resultSet = resultSet | set([document])
100

```

```

#for every relation in the relationSet, recursively call this method
102 #this way, every relation will eventually be added
for relation in relationSet:
104     print '[DEBUG] find_related_rechtspraak: for ' + str(relation) + ' in rel
        resultSet = resultSet | self.find_related_rechtspraak(relation, resultSet)
106
return resultSet
108

#####
110 ##          END OF MAIN ALGORITHM          ##
#####

```

10 Appendix D

DELFT UNIVERSITY OF TECHNOLOGY

BACHELOR EIND PROJECT

Oriëntatieverslag

Bas METMAN
Vincent VAN MIEGHEM

2 MEI 2014

1 Inleiding

Dit document beschrijft de voortgang in eerste twee weken, ook wel genoemd de oriëntatiefase, van ons Bachelor Eind Project voor PwC. We zullen een groot aantal onderwerpen beschrijven die wij hebben besproken in deze fase. Daarnaast geven wij de conclusies die we getrokken hebben na het bespreken.

1.1 Schets van het bedrijf

Wikipedia ¹: ‘PricewaterhouseCoopers, afgekort PwC, is een internationaal accountants- en belastingadviseursbedrijf, met hoofdzetel in Londen. PwC Nederland behaalde het in het boekjaar 2011-2012 een omzet van bijna 700 miljoen euro met een bezetting van bijna 4500 medewerkers.

PwC ontstond in 1998 door een fusie van Price Waterhouse en Coopers & Lybrand. De consultancytak werd in 2002 verkocht aan IBM. De advisory-afdeling werd echter opnieuw opgezet na de verkoop van de consultancytak. Tot en met 2010 stond het kantoor bekend als PriceWaterhouseCoopers. PwC wordt samen met KPMG, Ernst & Young en Deloitte tot de Big Four gerekend.

De formele naam is PricewaterhouseCoopers, aangevuld met een toevoeging van de betreffende entiteit (land of activiteit), maar vanaf 2010 maakt het bedrijf in haar commerciële uitingen gebruik van de naam PwC.’

Het project wordt uitgevoerd binnen de Business Unit HRS (Human Resource Services). In eerste instantie wordt het project gedaan voor de skillgroup EBTC (Employee Benefits and Tax Consultancy), oftewel loonbelasting. Later zal het misschien ook voor andere afdelingen ontwikkeld worden.

1.2 Achtergrond en aanleiding van de opdracht

De opdracht is tot stand gekomen omdat er vraag was naar een systeem dat in voorgedefiniëerde, gevalideerde bronnen kon zoeken. Ook is er vraag naar een zoekstelsel dat specifieke advies-functies bevat. Deze twee eisen zal het systeem aan voldoen. Daarnaast is er vraag naar het kunnen zoeken in verschillende interne archiveringsdatabases. Voor deze vraag zal dit systeem wat wij ontwikkelen een antwoord kunnen gaan bieden. Dit deel valt echter buiten de scope van dit project.

2 Opdracht

2.1 De opdrachtgever

De opdrachtgever van het project is Eric Dankkaart. Eric Dankkaart is partner bij PwC en houdt zich bezig met IT binnen de business unit Tax & HRS.

2.2 Probleemstelling

Google Search ² biedt niet genoeg en diepgaande zoekresultaten in databanken van de overheid. Daarnaast is er vraag naar een systeem dat ‘simpele’ vragen kan beantwoorden zoals ‘Geef me de loonbelastingtabbelen uit 2013’. Het systeem zou antwoord moeten geven door het weergegeven van de loonbelastingtabbelen uit 2013.

¹<http://nl.wikipedia.org/wiki/PricewaterhouseCoopers>

²www.google.com

2.3 Doelstelling

We verdelen het doel in twee delen:

- Een zoekstelsel wat grondig, efficiënt kan zoeken in zowel databanken (bestaand uit wetten en rechtspraak) van de overheid.
- Een zoekstelsel dat uitspraken van verschillende rechtbanken kan herkennen en een zekere hiërarchie kan weergeven, waarbij general language recognition gebruikt kan worden om met complexe queries data op te vragen.

2.4 Op te leveren producten

Een webapplicatie die makkelijk te bereiken is voor elke PwC professional die gebruik wilt maken van de applicatie. Intellectuele eigendommen en broncode zijn eigendom van PricewaterhouseCoopers Belastingadviseurs N.V..

3 Oriëntatie

In deze sectie zullen onze ondervindingen beschreven worden die we de eerste twee weken gedurende de oriëntatiefase hebben ondervonden. Tijdens deze fase hebben we uitvoerig met onze opdrachtgever gesproken en verschillende interviews gehouden met de uiteindelijke gebruikers van het stelsel.

3.1 Bronnen

De bronnen waarop we ons in eerste instantie gaan focussen om zoekdata uit te halen en die te indexeren zijn [overheid.nl](http://data.overheid.nl)³, [rechtspraak.nl](http://www.rechtspraak.nl)⁴, [belastingdienst.nl](http://www.belastingdienst.nl)⁵ en [loonheffing.nl](http://www.loonheffing.nl)⁶. Deze zijn onder te verdelen in drie typen:

1. Bronnen met een Open Data API ([overheid.nl](http://data.overheid.nl))
2. Bronnen met een databank te benaderen via deeplinks ([rechtspraak.nl](http://www.rechtspraak.nl))
3. HTML websites ([belastingdienst.nl](http://www.belastingdienst.nl) en [loonheffing.nl](http://www.loonheffing.nl))

De verschillende bronnen nemen verschillende formaten met zich mee, [overheid.nl](http://data.overheid.nl) en [rechtspraak.nl](http://www.rechtspraak.nl) leveren data in (verschillende soorten) XML formaat. [belastingdienst.nl](http://www.belastingdienst.nl) en [loonheffing.nl](http://www.loonheffing.nl) leveren hun data in HTML formaat.

Om deze formaten te kunnen indexeren in onze database, moeten ze uniform gemaakt worden. Wij zullen de XML en HTML bestanden daartoe parsen tot een uniform XML formaat, met tags die nuttige informatie bevat, zoals de titel en de bron van het document. Op deze manier kunnen wij de XML bestanden makkelijk inlezen in de database en efficiënt data opvragen.

³data.overheid.nl

⁴<http://www.rechtspraak.nl/Uitspraken-en-Registers/Uitspraken/Pages/default.aspx>

⁵<http://www.belastingdienst.nl/bibliotheek/handboeken/html/boeken/HL/>

⁶<http://www.loonheffing.nl/Loonheffing-MvM.htm>

3.2 Ontwikkel platform

Voor de ontwikkeling van de applicatie is gekozen voor het Django Framework ⁷. Dit om een aantal redenen:

1. Binnen PwC wordt .NET danwel Django gebruikt voor het ontwikkelen en uitrollen van webapplicaties.
2. Django is een van de meest gebruikte web platformen op dit moment, vanwege het feit dat er snel een website mee opgezet kan worden en makkelijk aan te passen is.
3. Voorkeur was voor een open source platform, waarbij er veel verschillende API's en libraries beschikbaar zijn. Dit geldt ook voor Django.
4. Django werkt via Haystack naadloos samen met Apache Solr, de search engine die gebruikt gaat worden.
5. Microsoft .NET is een te complex ontwikkelplatform voor het doel van de applicatie.

3.3 Search engine

Gekozen is voor Apache Solr ⁸, een search platform gebaseerd op de Java search library Apache Lucene ⁹. Solr is geschreven in Java, en kan worden gestart in een servlet container. Wij gebruiken hiervoor, in ieder geval in het begin, de Jetty servlet container ¹⁰. Wij hebben gekozen voor Solr omdat deze vaak gebruikt wordt voor webapplicaties, en de Lucene search library geeft snel resultaten terug. Wat vooral belangrijk is voor ons project, is dat er snel in tekst kan worden gezocht. Dit wordt ook ondersteunt door Solr. Verder is Solr samen met Haystack ¹¹ makkelijk te integreren in Django, wat voor ons handig is om snel een goede applicatie neer te zetten.

3.4 Gebruikers wensen

Na enkele interviews zijn we tot de volgende constatering gekomen:

1. Snelheid is enorm belangrijk. Gebruikers hebben geen zin om langer dan een seconde te wachten op zoek resultaten.
2. Gebruikers willen graag een chain zien van uitspraken van rechtbanken. Uitspraken die vooraf gingen aan een uitspraak van een hogere rechtbank moeten onder de laatste uitspraak te vinden zijn.
3. Gebruikers zouden graag zien dat het systeem gerelateerde zoektermen aan de ingevoerde zoekterm suggereert.
4. Het systeem moet de mogelijkheid geven om in bepaalde bronnen niet te zoeken.
5. Het moet zichtbaar zijn wanneer het systeem voor het laatste bronnen heeft opgehaald en bij elk resultaat een timestamp te vinden is.
6. Het systeem moet gebruiksvriendelijk en intuïtief zijn.

Deze wensen moeten uitgangspunten vormen bij het ontwerp en implementatie van het systeem.

⁷<https://www.djangoproject.com>

⁸<https://lucene.apache.org/solr/>

⁹<https://lucene.apache.org/>

¹⁰ <http://www.eclipse.org/jetty/>

¹¹<http://haystacksearch.org>

4 Planning

In deze sectie zal de aanpak en planning gedurende het project worden beschreven.

4.1 Aanpak

Het project zal bestaan uit een aantal fases. De eerste fase is de oriëntatiefase en loopt gedurende de eerste twee weken. De uitkomst van deze fase is beschreven in dit document. Daaropvolgend zal de implementatiefase plaatsvinden. Deze zullen we uitvoeren met behulp van de SCRUM methodiek, waarin we na elke 2 weken een werkend product leveren waarin elke keer nieuwe features zijn toegevoegd.

In de laatste week van juni zal het product worden opgeleverd en zal de eindpresentatie worden gegeven. Tussentijdse rapporten en documenten worden opgeleverd volgens het BEP schema.

Gedurende het project zullen we gemiddeld 2 dagen per week op het Rotterdams kantoor van PwC werken. De rest zal vanuit een locatie op de TU Delft gebeuren.

4.2 Versiebeheer

Voor het versie en issues beheer zal GitHub worden gebruikt. Deze manier van codebeheer is erg handig, omdat hiermee meteen logs worden bijgehouden en er makkelijk revisions teruggedraaid kunnen worden als er iets fout gaat. Issues aan de code en/of features kunnen hierin ook worden opgenomen en bijgehouden.

4.3 Planning

Week	Fase	
1	Oriëntatie	Gebruikerswensen vaststellen
2	Oriëntatie	Wensen analyseren en inlezen technologie
3	Implementatie	Opzetten basis functionaliteit
4	Implementatie	Opzetten basis functionaliteit
5	Implementatie	Opbouwen chaining functionaileit
6	Implementatie	Opbouwen general language functionaliteit
7	Implementatie	Uitbouwen general language functionaliteit
8	Implementatie	Uitbouwen general language functionaliteit
9	Testing	User acceptance testing
10	Presentatie	Presentatie en oplevering