University of Bergen

Department of Informatics

Developing data catalogue extensions for metadata harvesting in GIS

Author:
André Mossige

Long master thesis
June 2018

Acknowledgements

I would like to thank my supervisor Torill Hamre at the Nansen Environmental and Remote Sensing Center (NERSC) for providing great advice and assistance. She has always been helpful when I have encountered problems. I would also like to thank friends and peers from the study hall for incredible support. Finally, I want to thank my family for continuous encouragement and backing.

Abstract

Researchers in geoscience often use several Geographic Information Systems (GIS) to find and access different types of data for research use. The researchers do not always know in which GIS their needed data reside, and therefore might spend considerable amount of time searching for it. A better solution would be a GIS that combines the data in a single, searchable system. In this thesis we examine how a GIS that combines data from external data servers aid researchers in doing research. A GIS prototype with harvesting capabilities for a few commonly used data repositories in the geoscientific field is presented. First, we interview researchers to know about their GIS usage and problems, and assess relevant standards, protocols and technology to use in a GIS prototype. We present the prototype implementation, and demonstrate that it is quicker to use than searching several data repositories. The evaluation of the prototype show that the prototype has potential, but that improvements have to be considered, especially in regard to supporting harvesting from additional types of data repositories.

Contents

Acknowledgements **i**

Abstract **ii**

Contents **iii**

List of Acronyms **viii**

1 Introduction **1**

1.1 Background 1

1.2 Motivation for thesis 2

1.3 Overall and specific goals 2

1.3.1 Sub-goals 2

1.4 Research questions 3

1.5 Related work 3

1.6 Outline 4

2 Background **6**

2.1 Problem statement and analysis 6

2.1.1 Elaboration on goals 6

2.1.2 Elaboration on research questions 8

2.1.3 Requirements for the GIS prototype 8

2.2 Research methodology 9

2.3 Analysis of users 10

2.3.1 Use cases 10

2.3.2 Problems with current systems 12

3 Software stack 14

3.1 Definitions 14

3.1.1 Geographic Information System 14

3.1.2 History of GIS 15

3.1.3 GIS components 15

3.2 Standardisation organisations 17

3.2.1 Open Geospatial Consortium 17

3.2.2 Open Archives Initiative 17

3.3 Standards and protocols 17

3.3.1 Selection of standards 18

3.3.2 Metadata protocols 18

3.3.3 Data access 23

3.4 GIS alternatives 26

3.5 Comprehensive Knowledge Archive Network 27

3.5.1 Architecture 28

3.5.2 Extension mechanism 29

4 Implementation of GIS prototype 32

4.1 Development environment and methodology 32

4.2 CKAN default installation 32

4.2.1 Installation procedure 33

4.2.2 Base configuration 33

4.3 Enabling geospatial capabilities 35

4.3.1 Spatial extension 35

4.3.2 Harvesting extension 36

4.3.3 Geographic view extension 39

4.4 Experimenting with custom extensions 41

4.4.1 Generate and enable extension 41

4.4.2 Testing the extension 42

4.5 Customise existing OAI-PMH harvester 43

4.5.1 DIF metadata reader 44

4.5.2 Harvester overview 47

4.5.3 Data flow 49

4.5.4 Various smaller changes 51

4.6 Implementing TDS harvester 52

4.6.1 TDS crawler 52

4.6.2 Harvester overview 53

4.6.3 Data flow 54

4.7 Implementation challenges and solutions 56

5 System overview and demonstration 57

5.1 Finding data 57

5.2 Adding a new harvesting source 61

5.3 Filter data before retrieval 63

5.4 Download data 63

5.5 Summary 63

6 Evaluation **64**

6.1 Sub-goals 64

6.1.1 Overview of standards to build prototype 64

6.1.2 Types of data supported in prototype 65

6.1.3 GIS alternatives as prototype foundation 65

6.1.4 Set up a catalogue service 65

6.1.5 Implement harvesting capabilities 66

6.1.6 Implement simple geospatial visualisation 67

6.1.7 Evaluation of prototype 67

6.2 Research questions 68

6.3 Overall goal 69

7 Conclusion **70**

7.1 Summary of results 70

7.2 Future work 71

7.2.1 Perform user testing 71

7.2.2 Clean up harvested metadata for display 71

7.2.3 Improve display of TDS links harvested with OAI-PMH 72

7.2.4 Improve logic for harvesting periodic measurements 72

7.2.5 Display OPeNDAP access form directly in prototype 72

7.2.6 Improve detection of resource formats 73

7.2.7 Implement harvesting from more data sources 73

7.2.8 General improvements to harvesters 73

7.3 Conclusion 74

Appendices **75**

A Code listings **75**

A.1 IHarvester interface 75

A.2 CKAN configuration file 76

A.3 ThreddsHarvester implementation 81

Bibliography **84**

List of Acronyms

CGIS	Canada Geographic Information System.
CMIP	Coupled Model Intercomparison Project.
CMS	Content Management System.
CSW	Catalogue Service for the web.
DIF	Directory Interchange Format.
ESRI	Environmental Systems Research Institute.
GIS	Geographic Information System.
NERSC	Nansen Environmental and Remote Sensing Center.
NIRD	National Infrastructure for Research Data.
NMDC	Norwegian Marine Data Center.
OAI	Open Archives Initiative.
OAI-PMH	Open Archives Initiative Protocol for Metadata Harvesting.
OGC	Open Geospatial Consortium.
OPeNDAP	Open-source Project for a Network Data Access Protocol.
SDI	Spatial Data Infrastructure.
TDS	THREDDS Data Server.
WCS	Web Coverage Service.
WFS	Web Feature Service.
WMS	Web Map Service.
WWW	World Wide Web.

Chapter 1

Introduction

1.1 Background

When doing research in geographical sciences, geospatial data has to be analysed. Scientists have to find the data using a computer system, and it is usually downloaded before analysis. This is of often done through a Geographic Information System (GIS). Such system is an information system that integrates, stores, edits and, or shares geographic data.

Data used in geoscientific analysis and assessment originate from a wide range of sensors, algorithms and models. These sources often output their data with varied resolution in space and time, and different communities have developed their own formats and standards used to store it. The data is usually accessed through separate web sites or portals. For example a scientist may use a specific web site to find data about sea ice, and a different web site to find oceanographic data. The scientist then have to use both web sites to find all the needed data. Data is therefore sometimes hard to discover, as the user have to know where to look for it, and in what web sites to search.

In this thesis, we will investigate how we can harvest data from multiple sources using open-source frameworks and tooling. We will look at three frameworks to accomplish this task. The thesis will evaluate these frameworks and select the best one fulfilling a set of requirements.

In order for a GIS to allow easy discovery and harvesting of data, use of established standards is crucial. If standards are not used, two GIS will not know how to communicate with each other. For example a system can not know where or how to ask for data to harvest if a standard is not agreed upon. Such standards include, among others, DIF and ISO 19115 for metadata, NetCDF/CF for data, Solr, OAI-PMH and OGC CSW for search, and OGC WMS/WFS and OPeNDAP for data access.

1.2 Motivation for thesis

Researchers in geosciences use computer systems daily to aid their scientific work. Information systems are used for calculations and analyses, but also for searching and retrieving the data needed for those analyses. Data of different type and origin are often stored in distributed systems managed by different organisations with different APIs. For example sea ice data collected from the interior Arctic Ocean are stored on one data server, while sea ice data from the Svalbard region are stored on another server. The data may be stored in different formats, and the servers may have distinct APIs used to access them. Hence, a substantial number of the separate systems have to be searched when researchers look for data needed in their work. Additionally, the researchers have to be familiar with how all the systems user interfaces work to be able to use them effectively. Hence, searching for scientific data can be annoying, as it is not of direct value to the research.

This thesis examines the possibilities of harvesting metadata from different repositories into one. An optimal solution should gather all metadata automatically, and incorporate it into one searchable system. The result is a single system that researchers can use to find all the data they need.

1.3 Overall and specific goals

The main goal of the thesis is stated below. This main goal is split into smaller, more concrete sub-goals in the next section.

- Build a prototype GIS that acts as a data catalogue and harvests metadata from multiple sources, supporting the most commonly used, open standards.

1.3.1 Sub-goals

To complete the objective in Section 1.3, we split the overall goal into several concrete sub-goals.

1. Get an overview of the current relevant standards for building a GIS prototype.
2. Research types of data that should be supported in the prototype.
3. Assess relevant GIS frameworks to use as base for prototype.

4. Set up and configure a GIS as a catalogue service.
5. Implement support for relevant protocols and standards used for harvesting data.
6. Implement support for simple visualisation of geospatial data.
7. Evaluate the developed GIS prototype.

1.4 Research questions

The following research questions will help guide the thesis.

- How can a GIS that combines different data and formats within the system, aid a researcher in doing research?
- How should a systems software architecture be designed to allow extending its functionality?

1.5 Related work

Scientists in different application domains have studied the use of GIS frameworks as a foundation for developing data catalogues with capabilities for metadata harvesting. Scholz et al. (2017) [93] examines the use of a data catalogue to make big data available to the public for use in smart cities. They propose a prototype using Comprehensive Knowledge Archive Network (CKAN) for the data catalogue, and an extension to CKAN that harvests metadata from Hadoop Distributed File System (HDFS) [4]; distributed storage used for big data.

Fechner and Kray (2014) [83] explores possible tools to use to increase citizen engagement by providing interactive geo visualisations to the public. They also propose using CKAN for metadata storage, and using the Web Map Service (WMS) standard for visualisations.

A number of metadata models used in GIS frameworks are researched by Assaf et al. (2015) [74]. They compare and discuss the models and vocabularies used in the frameworks, and propose a unified metadata model. The model is based on many of the fields used in CKAN. The goal is to provide a unified metadata model for common use, to simplify communication between the GIS frameworks.

Similar to these studies, we will also propose a prototype that is based on an existing GIS framework, and we will extend its functionality to support our requirements. Different to the mentioned studies, we will focus on simplifying the discovery process of research data, a topic suitable for a master thesis.

1.6 Outline

We have presented the background and objectives for our work, and will now describe how we have structured the thesis.

Chapter 1 : Introduction

Introduces the general problem the thesis aims to solve. Gives objectives for a solution, and presents research questions guiding the thesis.

Chapter 2 : Background

Describes the objectives of the thesis in greater detail. Elaborates on the research questions from the introduction, and gives requirements for the GIS prototype. The research methodology used in the thesis is explained. The chapter is closed with an analysis of potential users of the prototype.

Chapter 3 : Software stack

Gives a technical description of definitions, standards and protocols to be used in the prototype. An assessment of GIS frameworks are given. The framework to be used in the prototype is explained in greater detail.

Chapter 4 : Implementation of GIS prototype

Details how we developed the prototype. The prototype foundation is set up and configured, and the implementation of two extensions are detailed.

Chapter 5 : System overview and demonstration

The prototype is demonstrated and use cases defined in Chapter 2 are examined. We will carry out the use cases and show the results.

Chapter 6 : Evaluation

An evaluation of the prototype is performed. Thesis objectives are compared with the results from the prototype. The research questions are answered.

Chapter 7 : Conclusion

We conclude the thesis work. We summarise the results from the prototype, and explore possible features to implement in future work. The final conclusion is given.

One appendix is also included.

Appendix A : Code listings

Presents code listings to illustrate the implementation of parts of our prototype.

Chapter 2

Background

In this chapter we will look at the use of GIS from the user perspective. We will define the different types of users relevant to this thesis. Some selected use cases are given, together with some of the user expressed problems with current GIS use.

Recognizing these researchers' perspectives will help us understand the context and the problems that we are trying to solve. In turn, we will build a prototype GIS that aims to solve the identified problems. These views are gathered from an informal interview with two researchers that uses GIS daily.

2.1 Problem statement and analysis

Before we go into detail on the users, we will elaborate on the goals and research questions from Chapter 1, and describe the requirements for the GIS prototype we will develop. We will then outline the research methodology used in the thesis.

2.1.1 Elaboration on goals

Below, we examine on the sub-goals given in Section 1.3.1 in detail.

1. **Get an overview of the current relevant standards for building a GIS prototype**

   The prototype we will build relies on communication with other GIS. Therefore it is necessary to know the relevant standards that are used in the communication between such systems. When developing our prototype we aim to use the most widely used standards in current GIS. To use these standards effectively we have to know which to use and how they work.

2. **Research types of data that should be supported in the prototype**

   To scope the amount of work required to build our prototype, we have to restrict the various types of data that should be supported. We will interview researchers and examine what types of data they use. Support for data formats typically used for these types should be implemented.

3. **Assess relevant GIS frameworks to use as base for prototype**

   Developing a fully featured GIS from scratch within the time frame of a master thesis is unfeasible. Therefore we will use an existing GIS framework as a base for our prototype. We have to assess the different alternatives available based on a set of requirements and pick the best one. Some general requirements are collectively decided with the thesis supervisor. For example the GIS framework should be open-source and widely used. More requirements appear as we interview GIS users in Section 2.3.

4. **Set up and configure a GIS as a catalogue service**

   We will install and configure the GIS framework picked from the previous sub-goal. It should support basic features needed in a catalogue service. For example it should be able to display datasets in a list, and show detailed information about each of them. The datasets in the catalogue should be searchable. For example typical search operations include searching based on geographic area and based on words that appear in the metadata of a dataset.

5. **Implement harvesting capabilities**

   We will extend the features of the basic catalogue service to support harvesting of metadata from external data servers. The harvesting should support collecting metadata with the standards and formats found in the first and second sub-goal.

6. **Implement simple visualisation of geospatial data**

   Simple visualisations of geospatial data should be supported. For example showing the layers of a geospatial dataset on a map within the browser. Standards and formats from the first and second sub-goal should be used.

7. **Evaluate the GIS prototype**

   When the GIS prototype is implemented, we evaluate the results, the development process, and conclude our findings.

2.1.2 Elaboration on research questions

In this section we will explain and motivate the research questions from Section 1.4. A research question is a question that identifies a problem that a thesis aims to solve [77]. The purpose of the research questions is to define the type of research we will be doing in the thesis, and to specify the thesis objectives. The questions will be answered in Chapter 6, after we have implemented and demonstrated our prototype.

- **How can a GIS that combines different data and formats within the system, aid a researcher in doing research?**

  We investigate whether combining data from different sensors into a single GIS is of help for researchers. Data originate from various sensors, models and algorithms, which often output their data with different resolution in time and space. The data may also have different representations, for example lines, sections or polygons. These different resolutions require that the datasets are stored using different types of formats, both in terms of resource format and metadata format. Developing a GIS that supports many different formats are some of the challenges in the thesis.

  Finding concrete solutions to how such GIS help researchers is crucial in making advancements in GIS research, as it would improve the daily work of researchers in geosciences. The improvements can likewise be applied to other GIS software.

- **How should a systems software architecture be designed to allow extending its functionality?**

  Computer system requirements differ for many users. However, many general requirements are similar for different users. When developing software it is often more efficient to start out with an existing foundation. This foundation already support the common, general requirements. The base is extended to support the specific requirements needed. To allow such an extension of functionality, the system need to have a flexible architecture. Finding out how such an architecture should be designed is important in developing software that can be used by as many users as possible, even with different requirements. Therefore, the same general functionality do not have to be implemented repeatedly.

2.1.3 Requirements for the GIS prototype

We describe the scope of functionality of the GIS prototype. The requirements will guide us in choosing a GIS framework to use as a base for the prototype. We aim at developing a prototype that fulfils as many of the requirements as possible.

- **Support most widely used standards and protocols used in geosciences**

The prototype should support standards and protocols related to both geographic resources and the metadata that describe them. If the prototype do not support the most used standards, we can not harvest external data sources, and users end up having to use other GIS to find all the data they need. The specific types of standards to support will be examined in Chapter 3.

- **Searching for datasets**

  Users should be able to search for the data they need. For example searching for words appearing in the metadata of the datasets, or searching for datasets within a certain geographic area. This will allow easier discovery of research data.

- **Harvesting capabilities for the most used standards**

  To be able to search within datasets from distributed data servers, the prototype have to be able to harvest metadata from those datasets. The standards that should be used to harvest the data are described Chapter 3.

- **Simple spatial visualisation**

  The prototype should be able to offer simple visualisation of geospatial data. This will allow users to see if a possible dataset contains the data they need, before they download it for use.

- **Downloading datasets**

  Users should be able to download datasets. The specific ways of how a user should be able to to download the datasets will be detailed in Chapter 3.

2.2 Research methodology

Peffers et al. (2007) [92] describes an effective research methodology for use in information systems research. The methodology contains six steps:

1. **Problem identification and motivation**: Identify a problem and justify why a solution is valuable.
2. **Definition of objectives for a solution**: Infer the objectives for a solution from the problem identification and determine what is possible.
3. **Design and development**: Determine the functionality of the artifact and implement it.
4. **Demonstration**: Illustrate how the artifact solves the identified problem.
5. **Evaluation**: Assess how well the artifact solves the problem.
6. **Communication**: Communicate the problem and solution to others.

The structure of the thesis loosely follows this process. We have in previous sections identified and motivated a problem, and given objectives for a solution. The next chapters will cover the design and implementation, demonstration and evaluation of our GIS prototype. Before we describe the development of our prototype, we will go into details of the potential users of our prototype. We will analyse their GIS usage and problems to be able to tailor our prototype to their needs.

2.3 Analysis of users

As we explained in Section 1.1, the scope of this thesis is related to the discovery of distributed geographic data, and harvesting its metadata to a centralised location. This way, a user can access the data originally hosted in different locations, through a single system.

We conducted an *informal* interview with two researchers from Nansen Environmental and Remote Sensing Center (NERSC). An informal interview is an interview where there is no predefined questions [27]. The advantage to such an interview is that the interview subject is freely able to give the answers they want, and the interviewer can ask follow up questions. Therefore the answers are not limited to those of an interview with prepared questions. Due to time constraints we were not able to interview more than two researchers. Having more interview subjects would give us a broader perspective, and therefore get a more realistic view of how researchers use GIS.

The goal of the interview was to examine how climate researchers use GIS, and to find out what problems, in their opinion, exist when working with these systems. During this interview we identified two categories of users related the thesis scope.

User category 1 Enters scientific data into data portals (data provider)

User category 2 Fetches and uses data in their analyses (data consumer)

The finished prototype from this thesis will fetch and use data that is already stored on a different web server. Therefore, we will not focus much on users of category 1, since the operations these users perform are not directly related to the thesis scope.

2.3.1 Use cases

In the interview we found several use cases that user category 2 performs on a regular basis. We describe the key use cases next.

Find and search for data

The main operation that user category 2 performs is searching for data. This may involve having to look through many different web sites or data portals, depending on what type of data is being looked for. For example, one of the users at NERSC mentioned that they use, among others, the following data sources:

- National Infrastructure for Research Data (NIRD) (former *NorStore*) [34] often used for accessing projections from climate models from Nordic countries
- Coupled Model Intercomparison Project (CMIP) [19] for global climate models
- NIRD for reanalysis data
- Norwegian Marine Data Center (NMDC) [33] for marine in situ (on site) data

Filter data before retrieval

The size of certain geographic dataset can be huge. One user reported that one NetCDF file can be several hundred gigabytes in size. Downloading such data can take a lot of time. Further, the user might not be interested in all parameters in the dataset. Therefore it might not be beneficial to download the complete dataset, and the user would like to somehow filter or restrict the dataset before download. The data can be restricted by entering some extra information. For example the user can specify the distinct layers they are interested in, or what time ranges should be included in the download, or what format is needed.

Some users would like to filter data within the dataset based on location, for example a dataset can contain data for the whole of Antarctica, but they might only be interested in a certain region of Antarctica.

Download data

The interviewed users reported that the data analysis is typically done in a separate desktop application with the data stored locally on their own computer. To achieve this the data has to be downloaded from the original location before it can be used by the researchers. The download is done by for example clicking a download button in a web page. A download of the selected dataset is then started in the browser. Another way this is achieved is by copying an URL from the GIS, and using the link in the desktop application that will allow the user to download datasets within it.

Analyse data in preferred application

When the data is downloaded locally, the researchers analyse the data in their preferred applications. This use case is not related to the scope of the thesis, and is therefore not discussed any further.

2.3.2 Problems with current systems

In this section we will cover some of the problems that the interviewed researchers at NERSC experienced. When these problems are identified and explained, we will have a better understanding of the specific problems, and can find solutions that will remove or reduce these problems in the prototype we will develop.

Having to use different web sites for different data

The main problem the users experienced when searching for data, was having to deal with several different web sites and data portals. When users need a specific type of data, they have to search on one web site, for example NIRD. When they need another type of data, they have to search on a different web site, for example CMIP.

The users often ended up having to look through many of the portals before finding the right data. The systems were not separated clearly, making it confusing to know in which system to search for what data. A small part of the problem was that the names of the data portals did not suggest what data can be found there. For example one user thought the names *CMIP Data Portal* and *NorStore* were not descriptive enough.

Being redirected before being able to download data

Another problem is that the user was redirected from one web site to one or several others before being allowed to download a dataset. Similarly to problem one, one web site may list metadata of datasets originally hosted elsewhere. Therefore, it may not be possible to download the dataset without navigating to the web site that hosts the actual data.

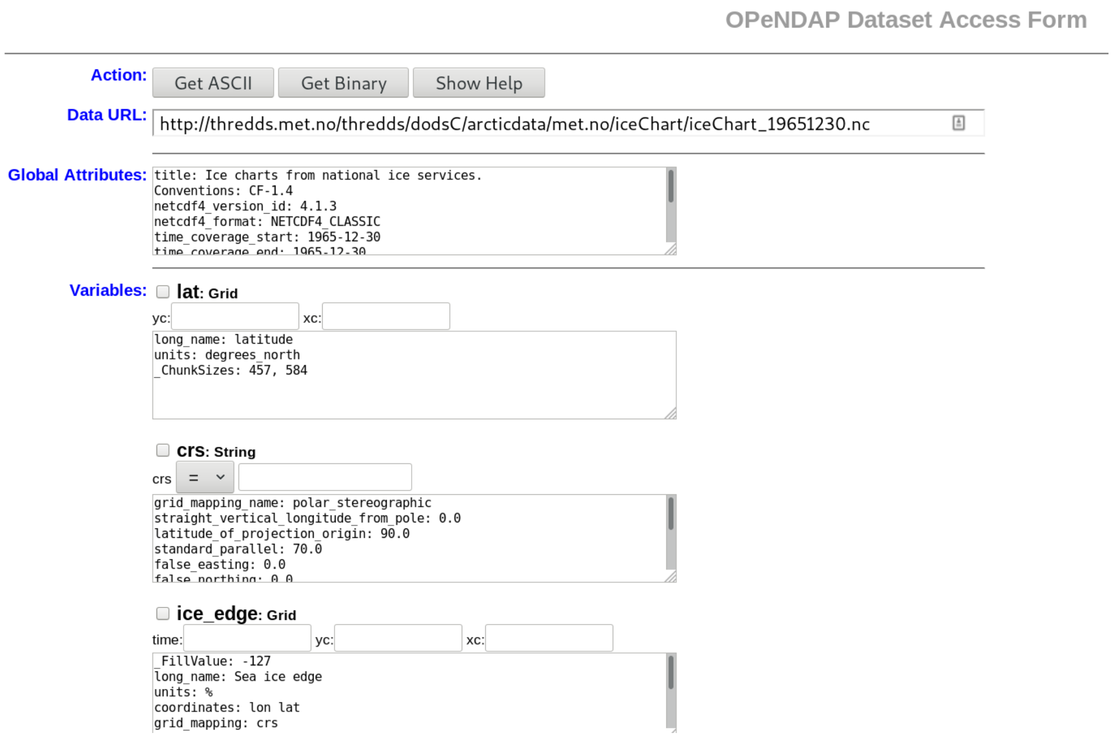
This results in the user having to adjust to and know how to use many different user interfaces. It makes it hard for the user to download data efficiently. The user interfaces often puts the various buttons and different options in different locations on the screen, making it more time consuming to get used to the various systems.

Verifying correct dataset before download

As mentioned in Section 2.3.1, one use case is about downloading a dataset before use. Since the size of the datasets often is large, the users want to be completely sure that they are downloading the datasets that they need.

Another use case is about the user restricting the dataset before download, to decrease its size. When filtering the data, the user is presented with many options. This is a helpful feature, but users report that it can be confusing at times, due to many options. Figure 2.1 shows such a user interface. Therefore, sometimes the user did not know if they had entered the options needed for the filtering they wanted. This sometimes resulted in too much of the dataset being downloaded, even if they thought they restricted it properly.

To solve these two problems the user would like a solution where you could do a simple visualisation to verify that the dataset one has found is the right one, and that it is filtered correctly. For example the user would specify the parameters needed, and the visualisation would update and reflect the change, allowing the user to verify correctness. Another advantage with such a visualisation is that the user can have a look at the different layers, and see which they need, instead of think which they need.



Data URL: http://thredds.met.no/thredds/dodsC/arcticdata/met.no/iceChart/iceChart_19651230.nc

Global Attributes:
```
title: Ice charts from national ice services.
Conventions: CF-1.4
netcdf4_version_id: 4.1.3
netcdf4_format: NETCDF4_CLASSIC
time_coverage_start: 1965-12-30
time_coverage_end: 1965-12-30
```

Variables:

lat: Grid

yc: xc:
```
long_name: latitude
units: degrees_north
_ChunkSizes: 457, 584
```

crs: String

crs =
```
grid_mapping_name: polar_stereographic
straight_vertical_longitude_from_pole: 0.0
latitude_of_projection_origin: 90.0
standard_parallel: 70.0
false_easting: 0.0
false_northing: 0.0
```

ice_edge: Grid

time: yc: xc:
```
_FillValue: -127
long_name: Sea ice edge
units: %
coordinates: lon lat
grid_mapping: crs
```

Figure 2.1: OPeNDAP Dataset Access Form

Chapter 3

Software stack

In Chapter 2 we investigated GIS usage from the perspective of users. To fully understand GIS, we also have to look at GIS and how they are built from a technical point of view. We will define GIS, give a short introduction to the history and explain the core components. Next, we will describe a set of standards and formats relevant to GIS prototype we will develop. Further, we will examine concrete frameworks suitable to build a GIS. When the technology is covered, we can in the next chapters go into detail about how one constructs a GIS with support for multiple data sources.

3.1 Definitions

3.1.1 Geographic Information System

A Geographic Information System (GIS) is a system that integrates, stores, edits and, or shares geographic data. A GIS accessed over the World Wide Web (WWW), is often called a Web-GIS. Such a system does not only constitute of the software itself, but also the hardware it is run on, the people using it, the organization in which the system is being used, and the geospatial data that is being used within it [78].

One use case for a GIS is map analysis. In such example, a researcher might put several maps on top of each other in layers for comparison. For example in urban planning a layer with a proposed road is put on top of a layer with the current construction, to see if it would fit. This thesis will focus on another use case, the integration and discovery part of a GIS: how researchers can effectively search and find the data they need for their analyses.

3.1.2 History of GIS

The evolution of GIS took place in the last half of the 20th century, with rapid advances in the later years. The first use of the term "Geographic Information System" appeared in academia in 1968 in a paper [96] by Roger Tomlinson. In this paper Tomlinson describes a GIS used to store, analyse and manipulate geographic data. The GIS allowed the use of overlays and measurements to determine the land capabilities in rural Canada. Tomlinson is in later years regarded as "the father of GIS". At the same time, in 1964, important theoretical concepts within spatial data handling was developed at Harvard by Howard T. Fisher [46].

In the 1970s, the development of the first publicly available GIS started. By the 1980s many vendors are involved in providing GIS software, for example Bentley Systems Incorporated with its CAD platform, and Environmental Systems Research Institute (ESRI). These GIS combined the early GIS features from Canada Geographic Information System (CGIS) and organizational features enabled by database structures [39].

Throughout the 1980s, desktop GIS applications appeared for DOS, and for Windows in the 1990s. These applications helped move GIS from research into commercial use. With the appearance of the internet, users start exploring possibilities of viewing GIS data over the Internet. This requires standardisation of data formats and data transfer, which has become a focus area in the later years. In the 21st century, many open-source GIS applications exist. Popular examples include *GeoServer*, *GeoNetwork* and *deegree*.

3.1.3 GIS components

Steiniger and Hunter (2011) [94] describe and analyse components of a Spatial Data Infrastructure (SDI). They conclude that free and open source software exist for all components required. The components needed in a SDI shown in Figure 3.1 and are briefly described below. In the next sections we examine the standards and protocols that the components are based on.

Data Service server serves spatial data and images. *GeoServer* is a popular web map server.

Spatial data storage stores the spatial data. This storage is often enabled by installing extensions to existing general databases. *PostgreSQL* [45] with the *PostGIS* [68] extensions is often used. A more lightweight alternative is *SQLite* [55] with the *SpatiaLite* [54] extension.

Catalogue registry provides services for the spatial metadata. For example adding, querying and display of metadata. *CKAN* [2] and *deegree* [20] are alternatives.

GIS processing server exposes spatial processing functionality. For example map analysis and transformations. The functionality can be provided by *GeoServer* [26] with the *OGC Web Processing Service (WPS)* [35] standard.

Client used by users for creating, updating (*data provider*) and analysis of spatial data. Desktop clients (thick clients) often provide more functionality than browser-based clients (thin) used for viewing and querying spatial data.

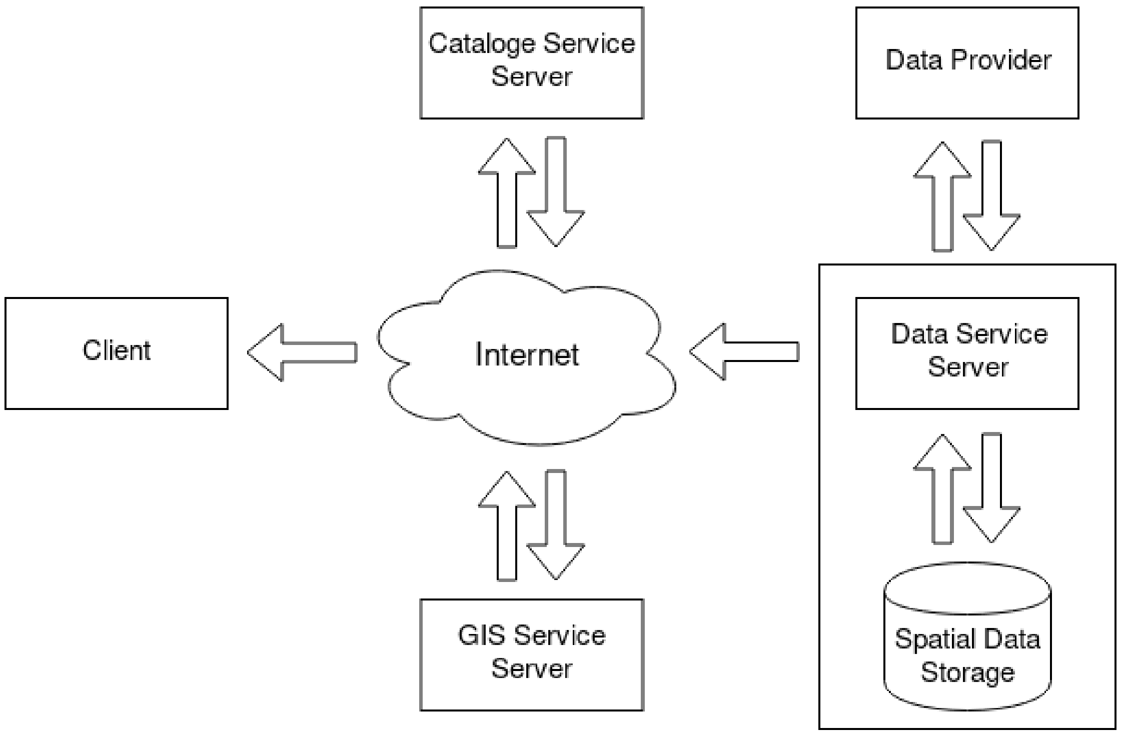


Figure 3.1: Components of a Spatial Data Infrastructure (adapted from [94])

When using a GIS, many or all of the components communicate together to provide the action a user requests. For example when users search for data, they use their client to access a catalogue service over the internet. This catalogue service may contain metadata records about geographic resources harvested from external data servers. When the user have found the dataset they want to access, the client requests the data from the data service, which queries the spatial data storage and returns the data. If the user want to do some analysis on the data, the client might request specific data transformation actions from a GIS service, which returns the new resource.

3.2 Standardisation organisations

Some of the technical standards we will use in the prototype are created by standardisation organisations. The primary goal of these organisations are to coordinate the creation of technical standards. We will give a short introduction to two big standardisation organisations.

3.2.1 Open Geospatial Consortium

The Open Geospatial Consortium (OGC) is an organization consisting of over 500 international companies, government agencies and universities. The members cooperate to create open standards for use in the geospatial community. By collaborating on open standards, the OGC hope to achieve better interoperability between geospatial services. When several development teams work on independent online services, the services will be able to work together. This will in turn increase the possibilities to create complex spatial information systems [88]. For example, using the Catalogue Service for the web (CSW) standard [79], a GIS can expose metadata about its spatial data for clients to use. Without such a standard, clients would not be able to know how to fetch and use the metadata.

3.2.2 Open Archives Initiative

The Open Archives Initiative (OAI) [36] is an association whose goal is to promote and develop open interoperability standards. They want to improve access to archives that contain digital content. Through the OAI interoperability framework, service providers can expose metadata about their data and services, allowing clients to access and harvest it freely. In Section 3.3.2 we describe the interoperability framework.

The main work of OAI was aimed at the E-Print community, exposing metadata about research papers and scholarly work, making them more accessible [37]. However, the technological foundation of the interoperability framework is independent of content type. Therefore the framework is applicable to metadata of all digital data.

3.3 Standards and protocols

Standards and protocols can be used to communicate with data repositories. Some of the standards are used to access *metadata* describing some resources, while other standards are used to access actual datasets in a repository. The prototype we will develop would not be able to harvest metadata from multiple data sources if a common standard was not agreed on. The standards relevant to this thesis are detailed in the following sections.

3.3.1 Selection of standards

Recall some of the data repositories being used by researchers from NERSC from Section 2.3.1: NIRD, CMIP and NMDC. By researching the different data sources, we discovered various methods to be able to access data from them. NMDC provide metadata access for example via Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH). Much of the data in the mentioned CMIP repository are originally stored on a THREDDS Data Server (TDS). Instead of harvesting CMIP, we will harvest the data directly provided by the TDS. Unfortunately, it does not seem that NIRD provide an easy way to access its data. NIRD do provide open data published in scientific articles, but access is done by sending a download link by e-mail. We will therefore focus on harvesting from the two other data repositories, using OAI-PMH and TDS. In the next sections we describe the standards and protocols potentially needed to be able to harvest and access data from these sources.

3.3.2 Metadata protocols

Metadata is defined as "data that describe other data" [32]. Geographic metadata is data that describe other data in the context of geography. For example the metadata can contain information about the author of a particular dataset, when the dataset was created, and a short description of the dataset.

For this thesis, the important fields of the metadata is about where the data is hosted and can be accessed, its format, and what protocol should be used for the access. In the next sections we will cover the metadata formats and standards that will be used in the implementation of our prototype.

Directory Interchange Format

The Directory Interchange Format (DIF) is a metadata standard that is used to describe earth science datasets for communication between information systems [89]. It contains several mandatory and optional fields. However, as many of the optional fields should be included to increase the understanding of the dataset. For example DIF can describe parent and child relationships between resources in a dataset. The DIF fields are explained in [21]. Some of the most used ones are listed below.

- Entry_ID: A unique document identifier of the metadata record.

- Entry_Title: The title of the dataset.
- Summary: Composed of a required `Abstract` field, and an optional `Purpose` field. They briefly describe the dataset and its purpose.
- Metadata_Name: Identifies the current DIF standard. This field is often automatically generated by GIS software.
- Data_Center: Contains many sub-fields detailing the data center, organisation or institution responsible for the dataset.
- Parameters: A set of keywords that represents the dataset. This field is used when searching and finding datasets in a big collection. The parameters field must contain several sub-field: *category*, *topic* and *term*. These fields describe the category and topic of the dataset. The values of these fields are often taken from a vocabulary. For example values can be *Oceans*, *Land Surface* or *Paleoclimate*, among others.
- Related_URL: Contains URL links pointing to additional information about the data. For example this field can contain links to access the dataset. A sub-field *URL_Content_Type* must be included that describes the link type. For example the link type can be *GET DATA*, describing that the link points to an Open-source Project for a Network Data Access Protocol (OPeNDAP) service.

Dublin Core

Dublin Core [23] is a metadata format for describing digital resources [64]. The original specification contains 15 fields. Some of them are *title*, *creator*, *subject* and *description*. Dublin Core is less suited for describing geographic resources since it is more generalised than DIF. An example of the Dublin Core metadata format is shown in Listing 3.1. Dublin Core is not extensively used in this thesis, and we will not go into more details about this standard.

```
<oai_dc:dc xsi:schemaLocation="http://www.openarchives.org/OAI/2.0/oai_dc↩
    /
    http://www.openarchives.org/OAI/2.0/oai_dc.xsd">
    <dc:identifier>
        http://thredds.met.no/thredds/catalog/arome5/catalog.html
    </dc:identifier>
    <dc:title>AROME METCOOP 0.5 km</dc:title>
    <dc:description>
        This is the 2m air temperature generated by AROME 0.5 km and post
        processed. This is a rolling archive where only the most recent
        forecast is shown. In order to access historical data, please
        contact the institute.

        This is a preliminary version of the dataset and further ↩
            information
        will be provided in subsequent metadata releases. These metadata ↩
            are
        currently under development.
    </dc:description>
    <dc:creator>Norwegian Meteorological Institute (met.no)</dc:creator>
    <dc:coverage>2015−06−18 to </dc:coverage>
    <dc:rights>CC BY/NLOD</dc:rights>
    <dc:coverage>FIXME</dc:coverage>
    <dc:subject>
        Atmosphere > Atmospheric Temperature >
        Surface Temperature > Air Temperature
    </dc:subject>
    <dc:subject>climatologyMeteorologyAtmosphere</dc:subject>
</oai_dc:dc>
```

Listing 3.1: Example of Dublin Core metadata for air temperature measurements

Catalogue Service for the Web

Catalogue Service for the web (CSW) is one part of the *Catalogue Services* standards from OGC [79]. The standard defines how a GIS should expose a catalogue of its metadata records, services and other resources. For example a GIS can expose the spatial data it has stored, and what service can be used to access it. If the GIS exposes a service such as WMS, detailed in Section 3.3.3, a client can use this information to get a preview of the geospatial data for own use.

CSW defines several operations [25, 90]. These operations allow a client to query information from a CSW server that will return an XML response with the requested data. The mandatory operations are:

GetCapabilities lets a client ask for information about a service, including what services it can provide and where the client can access those services. For example a response might include that the server supports *GetRecords* and *GetCapabilities*.

DescribeRecord is used to query type information about the information model provided by a service. For example a client can ask for information about the model type *Record*, and receive an XML schema definition for the *Record* type.

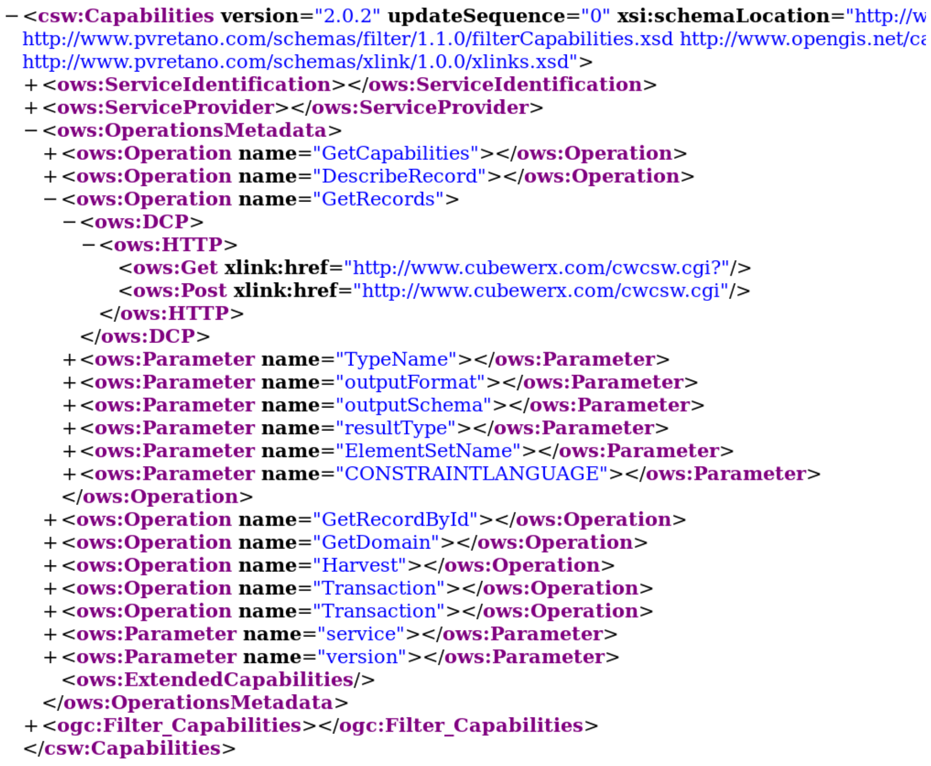
GetRecords performs a search for several records. The client can pass two parameters, `typeName` and `Constraint`, to specify what types of entities should be searched for, and what constraints should be applied. For example a constraint can be that a provided string should be included in a specified field in a record.

GetRecordById lets a client search for a record by ID. For example the *GetRecords* operation might return with references to other records. *GetRecordById* can then be used to query these records.

Three optional operations exist: *GetDomain*, *Harvest* and *Transaction*. The first is used to query runtime information about request parameters. For example a client can use this operation to discover the allowed values for a specific parameter. The *Harvest* operation is used to tell a CSW server to create or update metadata records by harvesting them from an external location. The *Transaction* operation is used to create or update records. This time a metadata record is not pulled into the system by the CSW server itself, but it is rather pushed to it by a client.

A sample *GetCapabilities* request is shown below. This example queries a service named "CSW" at the `http://localhost:8080/geonetwork/srv/eng/csw` URL. Parts of a typical response is shown in Figure 3.2. Note the operations the CSW server supports, and the URL endpoints pointing to them.

```
http://localhost:8080/geonetwork/srv/eng/csw?request=GetCapabilities&
   service=CSW&acceptVersions=2.0.2&acceptFormats=application%2Fxml
```



```
- <csw:Capabilities version="2.0.2" updateSequence="0" xsi:schemaLocation="http://w
  http://www.pvretano.com/schemas/filter/1.1.0/filterCapabilities.xsd http://www.opengis.net/ca
  http://www.pvretano.com/schemas/xlink/1.0.0/xlinks.xsd">
  + <ows:ServiceIdentification></ows:ServiceIdentification>
  + <ows:ServiceProvider></ows:ServiceProvider>
  - <ows:OperationsMetadata>
    + <ows:Operation name="GetCapabilities"></ows:Operation>
    + <ows:Operation name="DescribeRecord"></ows:Operation>
    - <ows:Operation name="GetRecords">
      - <ows:DCP>
        - <ows:HTTP>
            <ows:Get xlink:href="http://www.cubewerx.com/cwcsw.cgi?"/>
            <ows:Post xlink:href="http://www.cubewerx.com/cwcsw.cgi"/>
          </ows:HTTP>
        </ows:DCP>
      + <ows:Parameter name="TypeName"></ows:Parameter>
      + <ows:Parameter name="outputFormat"></ows:Parameter>
      + <ows:Parameter name="outputSchema"></ows:Parameter>
      + <ows:Parameter name="resultType"></ows:Parameter>
      + <ows:Parameter name="ElementSetName"></ows:Parameter>
      + <ows:Parameter name="CONSTRAINTLANGUAGE"></ows:Parameter>
      </ows:Operation>
    + <ows:Operation name="GetRecordById"></ows:Operation>
    + <ows:Operation name="GetDomain"></ows:Operation>
    + <ows:Operation name="Harvest"></ows:Operation>
    + <ows:Operation name="Transaction"></ows:Operation>
    + <ows:Operation name="Transaction"></ows:Operation>
    + <ows:Parameter name="service"></ows:Parameter>
    + <ows:Parameter name="version"></ows:Parameter>
      <ows:ExtendedCapabilities/>
    </ows:OperationsMetadata>
  + <ogc:Filter_Capabilities></ogc:Filter_Capabilities>
  </csw:Capabilities>
```

Figure 3.2: Example of CSW GetCapabilities response listing supported operations

Open Archives Initiative Protocol for Metadata Harvesting

Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) is a protocol for harvesting *metadata* descriptions into a single archive [38]. This allows us to build a service based on aggregated metadata from a collection of archives. Actual data resources are not harvested with OAI-PMH or CSW, only metadata describing it. OAI-PMH supports various metadata schemas. The most common is Dublin Core (Section 3.3.2). OAI-PMH has no opinion about the type of resource the metadata describe. For example OAI-PMH can harvest metadata about sea ice or weather forecasts in different formats. Similar to CSW, OAI-PMH operates through HTTP with responses in XML.

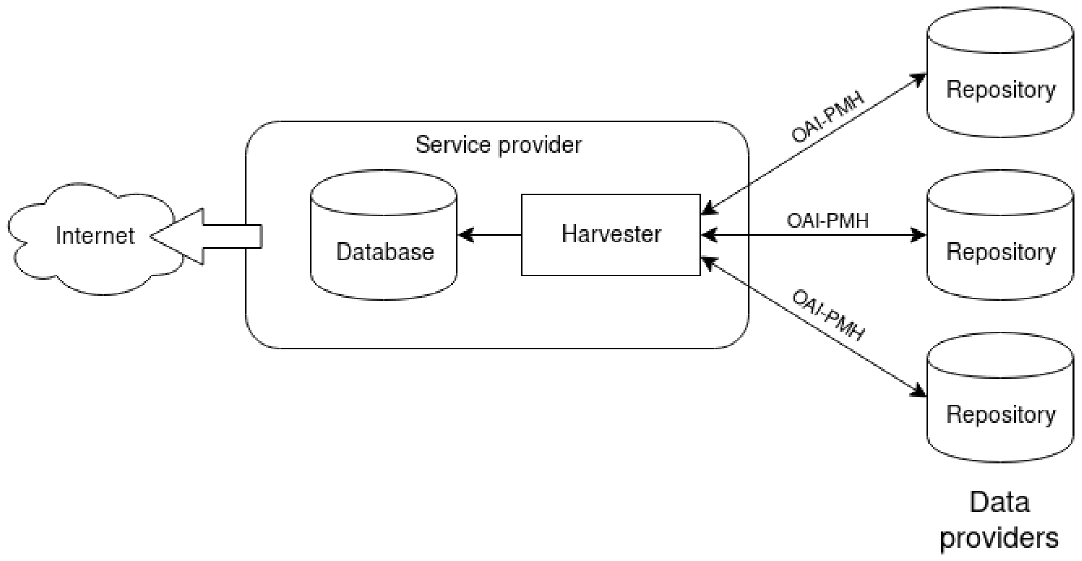


Figure 3.3: OAI-PMH protocol overview

Figure 3.3 displays an overview of the OAI-PMH protocol. Data providers hold repositories containing metadata records about some resources. A harvester collects the records from the providers, using the OAI-PMH protocol. The harvester queries the providers for their resources, and they return the metadata. The records are stored in a database that can be exposed through a *service provider* that is available to clients and other users on the internet.

The harvester is responsible for several important decisions. For example how often it should harvest data from a repository, or what happens when a resource is deleted. The harvester should also not reharvest resources that are already stored in the database. Harvesters should implement this functionality differently based on types of data stored in the repositories.

OAI-PMH defines several *verbs* (operations) that are used to make requests to OAI-

PMH repositories [87]. The operations are similar to those of CSW. They are briefly described below (in alphabetical order).

GetRecord is used to retrieve a single metadata record based on resource identifier.

Identify is used to get information about a repository. For example information about the supported OAI-PMH version and contact information.

ListIdentifier returns the identifiers that can be harvested from a repository.

ListMetadataFormats lists the available metadata format supported by a repository. For example Dublin Core and DIF.

ListRecords is used to harvest several records from a repository based on resource identifiers. Typically the `ListIdentifier` operation is used to gather the identifiers first, and then harvest the records with `ListRecords`.

ListSets is used to get information about the structure of a repository.

3.3.3 Data access

Web Map Service

While CSW and OAI-PMH is related to accessing *metadata* about resources, Web Map Service (WMS) is related to the access of *raster images* of the actual data that a GIS provides [80]. The protocol includes an interface specifying how the data should be requested. When some data is requested, a raster image is returned, for example in *JPEG* or *PNG* format.

The client that requests the data can provide some parameters in the request, describing what data should be returned. For example the client can restrict the geographic area of the image, which map layers should be included, or the alpha value of the image (useful for combining layers). WMS include two mandatory operations, and one optional:

GetCapabilities Similar to CSW, this operation return information about the WMS. For example supported map image formats, what metadata format versions are supported, available map layers and information about them.

GetMap This operation returns the actual map image. The client can include parameters to specify the data query. For example map width and height, type of coordinate reference system, rendering style and image format.

GetFeatureInfo (optional) Allows a client to request additional information about features in the returned map. For example a user might have returned a raster image using `GetMap`. The user then clicks on a specific location on the image, and the `GetFeatureInfo` can return information about what is located at that area.

An example of a `GetMap` request is shown below. For example the request specifies the `GetMap` operation, that the returned image format should be PNG, and that the image should be 500 pixels in width and 250 pixels in height. A possible response image is shown in Figure 3.4.

```
https://webmap.ornl.gov/ogcbroker/wms?SERVICE=WMS&VERSION=1.1.1&REQUEST=GetMap&
LAYERS=980_13&FORMAT=image/png&STYLES=default&TRANSPARENT=true&SRS=EPSG:4326&
BBOX=-180,-90,180,90&WIDTH=500&HEIGHT=250&TIME=1990-12&EXCEPTIONS=application/
vnd.ogc.se_xml&originator=SDAT
```

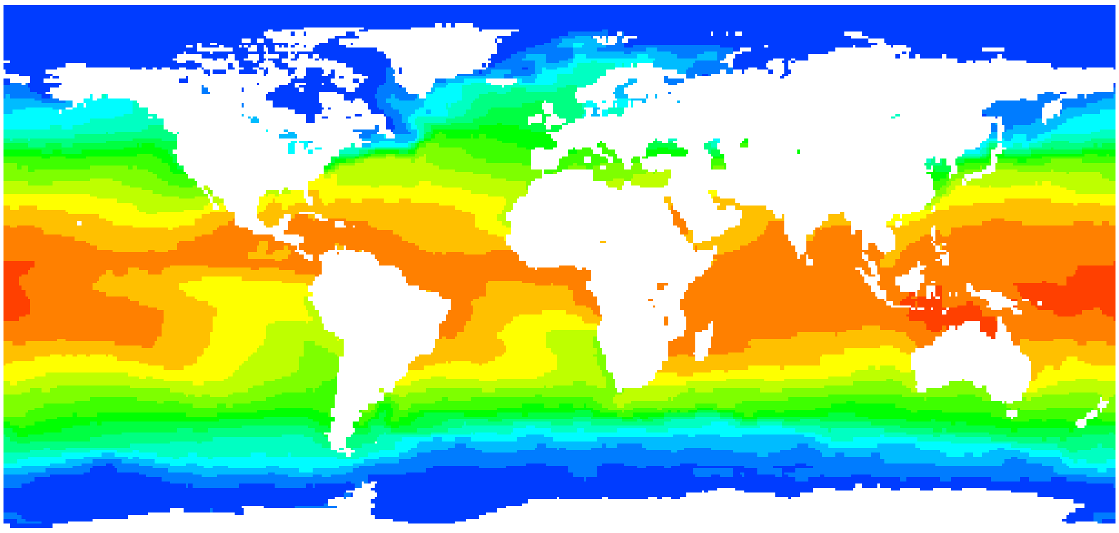


Figure 3.4: Example WMS response

Open-source Project for a Network Data Access Protocol

OPeNDAP is a protocol for accessing remote, distributed research data [49]. OPeNDAP includes both server and client software, used for storage and access. The size of geospatial data is often considerable, and one is not always interested in the whole dataset (Section 2.3.1). OPeNDAP solves this problem by sub-sampling the data using constrained queries. Thus, the user can limit the data they are returned.

The data is stored at the server in binary form, that can be accessed via a client over HTTP. The data is transferred by default in the original binary format. The server also supports transmitting data in NetCDF, GeoTIFF, JPEG2000, JSON and ASCII format.

Several clients exist for OPeNDAP. For example access via command line or data analysis packages like *Ferret* [24]. For this thesis, the relevant client is the browser. By adding

``.html'' to the OPeNDAP URL, the user accesses the *OPeNDAP Server Dataset Access Form*. Figure 2.1 shows a number of options to construct our access query. The options are explained below.

Actions When we have added our constraints we can use these actions to request the output format we need. Formats supported include ASCII and binary.

Data URL The data URL is the URL used to access the data. As we update our query constraints, the data URL is updated.

Global Attributes These attributes are used for reference. They apply for the whole dataset.

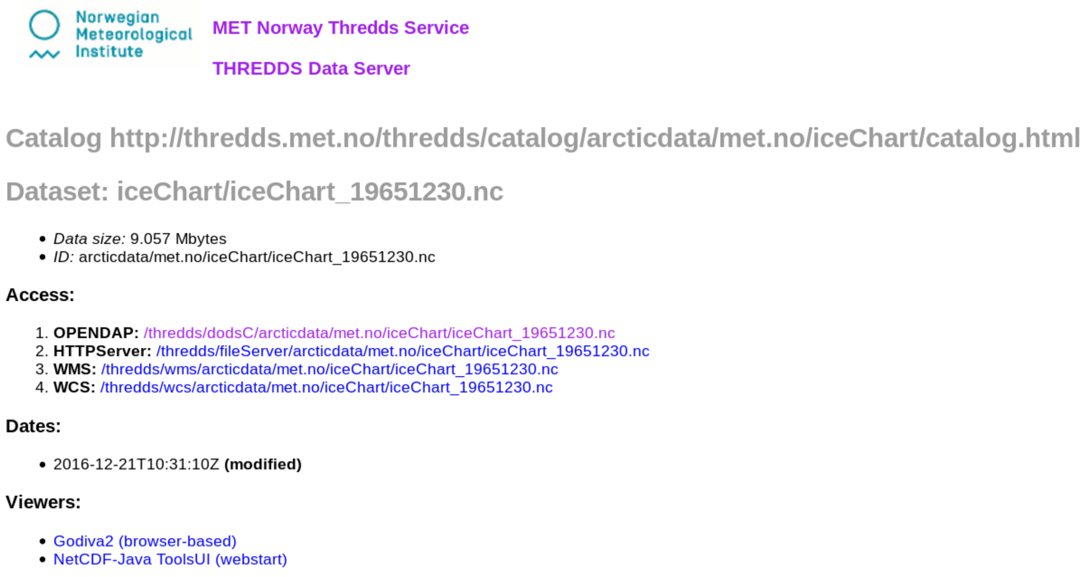
Variables Last, the dataset variables are listed. We can enter values for each variable which will constrain the output dataset. By checking a check box, we can decide to completely remove variables that we are not interested in from the dataset.

THREDDS Data Server

THREDDS Data Server (TDS) is a web server written in Java that provide both data access and metadata about scientific data [72]. The web server supports a number of the standards and protocols used in the scientific communities. Access is provided via for example OPeNDAP, Web Coverage Service (WCS) [75] for gridded data or access through HTTP. Visualisation is supported by default via WMS.

One of the advantages of TDS is that it can automatically generate a *THREDDS Catalogue* based on the contents of the database [59]. This catalogue encodes the data in XML, where the dataset metadata can be placed. Thus, clients can access the catalogue to find out what data is available for them to consume, and to determine how to access it. The XML catalogue can be automatically transcoded into an HTML web site, allowing it to be easily displayed to a user. The HTML catalogue is shown in Figure 3.5.

Many sensors generate measurements with regular intervals. Each of these measurements may constitute one data point in a database, and will thus be seen as a single entry in a catalogue. To avoid cluttering the catalogue, TDS have the ability to create virtual aggregations of such periodical measurements. Researchers can then access this single aggregated dataset, and constrain the dataset using OPeNDAP. This achieves better usability since the user do not have to manually scroll the list of possibly hundreds of datasets.



MET Norway Thredds Service

THREDDS Data Server

Catalog http://thredds.met.no/thredds/catalog/arcticdata/met.no/iceChart/catalog.html

Dataset: iceChart/iceChart_19651230.nc

- *Data size:* 9.057 Mbytes
- *ID:* arcticdata/met.no/iceChart/iceChart_19651230.nc

Access:

1. **OPENDAP:** /thredds/dodsC/arcticdata/met.no/iceChart/iceChart_19651230.nc
2. **HTTPServer:** /thredds/fileServer/arcticdata/met.no/iceChart/iceChart_19651230.nc
3. **WMS:** /thredds/wms/arcticdata/met.no/iceChart/iceChart_19651230.nc
4. **WCS:** /thredds/wcs/arcticdata/met.no/iceChart/iceChart_19651230.nc

Dates:

- 2016-12-21T10:31:10Z **(modified)**

Viewers:

- Godiva2 (browser-based)
- NetCDF-Java ToolsUI (webstart)

Figure 3.5: THREDDS Catalogue

3.4 GIS alternatives

We will now examine existing GIS solutions to use as a base for our prototype. The alternatives should support as many of the requirements given in Section 2.1.3. Otherwise, the GIS should have a flexible design. This will allow us to use the GIS as a base, and extend and implement the missing features that are required.

Amorim et al. (2017) [73] have compared GIS for research data management. The comparison consider architecture, flexible metadata and interoperability of the various GIS. They list several GIS suitable for research data management. Of those mentioned, *ePrints*, *DSpace* and *CKAN* are the most popular ones, and are therefore considered relevant alternatives to our prototype.

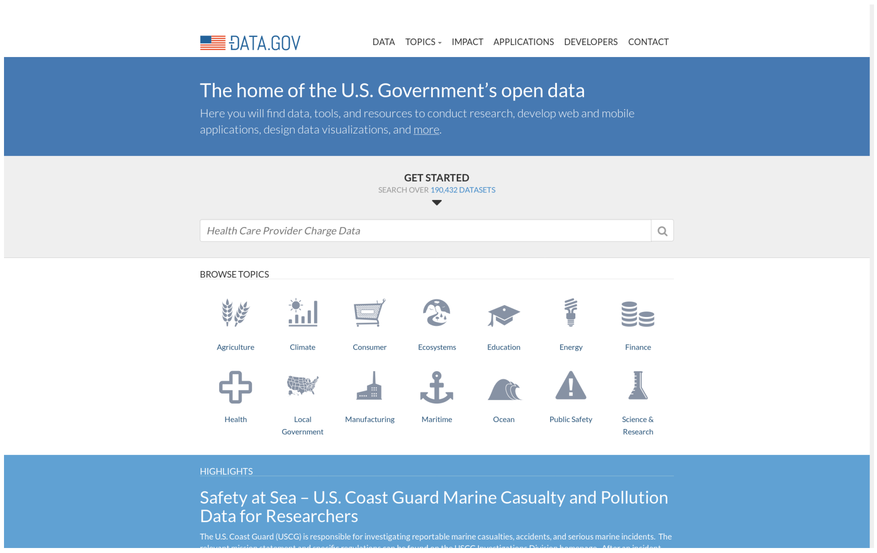
According to Amorim et al., these GIS have similar advantages. They are open-source, are easy to install, and are widely used within both research and governmental institutions. The systems are flexible, allowing for customizations. However, “due to its complex architecture, DSpace may require a higher level of expertise when dealing with custom features” [73]. The systems have various default support for open standards and protocols. Amorim et al. concludes that CKAN may be advantageous to the other alternatives. We will therefore use CKAN as the foundation for our prototype.

3.5 Comprehensive Knowledge Archive Network

Comprehensive Knowledge Archive Network (CKAN) is an open-source Python based data management system. CKANs purpose is to aid in the process of sharing, publishing, finding and using data [2]. CKAN support storage of simple data, for example HTML, CSV and JSON. Bigger data resources are not hosted in CKAN, for example several terabytes big NetCDF files are not appropriate. In these cases its metadata is stored instead.

The metadata stored do not have to be manually entered into the CKAN instance itself, but can be *harvested* from other sources or other CKAN instances. The harvesting is enabled by CSW, OAI-PMH and TDS, explained in Section 3.3.2. Thus, CKAN is a GIS that can be used to solve the problem about standardizing how data is discovered. Because data from many different sources are discoverable and searchable from elsewhere than it is originally hosted, the user do not have to search multiple data sources before the required data is found.

As mentioned, CKAN is open-source. This has many added benefits. For example, CKAN has an active developer and user community. The people behind the community provide great support, through mailing lists [8] and GitHub issues [14]. CKAN is also used by many governmental institutions, as a web page to find open data. For example `www.data.gov` is created with CKAN. Figure 3.6 illustrates some of what can be achieved. The home page lets the user view data for different topics, and download the data needed. Free text searching and display of the latest latest added record is also supported.



The home of the U.S. Government's open data

Here you will find data, tools, and resources to conduct research, develop web and mobile applications, design data visualizations, and more.

GET STARTED

SEARCH OVER 190,432 DATASETS

Health Care Provider Charge Data

BROWSE TOPICS

HIGHLIGHTS

Safety at Sea – U.S. Coast Guard Marine Casualty and Pollution Data for Researchers

The U.S. Coast Guard (USCG) is responsible for investigating reportable marine casualties, accidents, and serious marine incidents. The

Figure 3.6: U.S. Government open data homepage

3.5.1 Architecture

One of CKANs greatest advantages is its modular architecture. The design is displayed in Figure 3.7. Routes, views, business logic, models and API are split according to the concept of separation of concerns, in a layered architecture [7]. The main components are described below. Only the components relevant to this thesis are included.

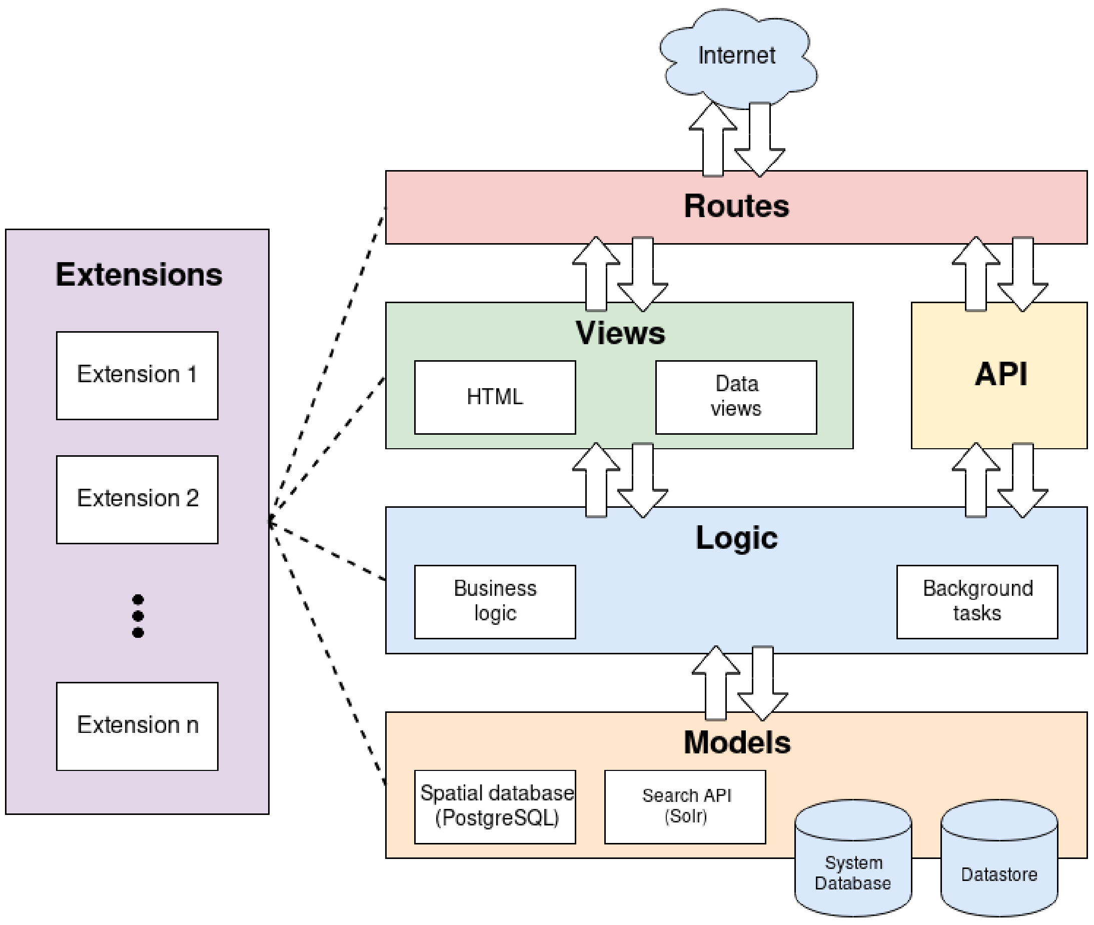


Figure 3.7: CKAN architecture (adapted from [7])

Routes The routes define the relationships between URLs and their corresponding views. For example when a client visits `http://<hostname>/harvest`, the router determines that the harvesting user interface should be rendered.

Views The views handle requests and serve a response by rendering the HTML to the user client. Access checks can be used to define what views or parts of views should be shown to users that have different access rights. Extensions can create their own views, and can incorporate them wherever needed.

Logic The logic includes the business logic and background tasks in CKAN. For example the business logic include a method for creating a dataset entry that will be displayed in the list of datasets. We will see in later chapters that we create a harvester extension that calls this method (via the API) to create datasets of the harvested data.

Models The model layer contains the data that is stored in CKAN. For example the spatial metadata is stored in a PostgreSQL database, and it is indexed by Solr. Extensions can access the model layer to query the database and retrieve needed data.

API The API provides a way of programmatically accessing the CKAN logic externally, outside of CKAN. For example we can list all datasets stored within from the API, and use for other purposes.

Extensions Separate extensions can be installed to customize and extend CKANs features. Extensions can hook into and modify all parts of CKAN. This extension mechanism is explained in detail in Section 3.5.2.

3.5.2 Extension mechanism

CKAN does not provide many advanced geospatial features by default, as its base installation is simple. Many of the required features are provided by extensions. Currently over 200 CKAN extensions have been created by the community. These extend the default features of CKAN [12]. For example one extension adds the possibility to do location search within geospatial data, while another adds features to harvest metadata from other CKAN instances. In this section we will examine how CKAN extensions work in general.

Plugin interfaces

To make an extension modify the standard functionality of CKAN, it has to implement one or several of the CKAN *plugin interfaces* [42]. The CKAN core will call the implementation of these interfaces, therefore changing its functionality [69]. For example the interface `IRoutes` contains methods for modifying the CKAN routes, letting extensions create new web pages with new content. Relevant interfaces for our GIS prototype include `IHarvester`, `IResourceView` and possibly others. These will let us implement custom harvesters, and create view renderings for the data that we harvest.

Plugin toolkit

Another alternative for extensions to get access to the CKAN core is to use the *plugin toolkit* [44, 6]. The plugin toolkit is a Python module containing methods, classes and exceptions from the CKAN core. It provides a `get_action()` method that extensions can use to call internal methods from the CKAN Action API [3]. The Action API is an API that exposes CKANs core functionality for use by clients and extensions. The methods returned by `get_action()` are considered safe to use, as they are backwards compatible with earlier CKAN versions.

Exception handling

The plugin toolkit also provides exceptions to use for error handling [5]. For example if an extension tries to call a method from the Action API that it is not authorised to use, it can be handled by exceptions from the plugin toolkit. An example of exception handling is shown in Listing 3.2. If a user without administrator rights try to access the list of all members, an exception is thrown, and an explanatory message is returned.

```
try:
    members = toolkit.get_action('member_list')(
        data_dict={'id': 'curators', 'object_type': 'user'})
except toolkit.ObjectNotFound:
    # The curators group doesn't exist.
    return {'success': False,
            'msg': "The curators groups doesn't exist, so only sysadmins ↩
                "
            "are authorized to create groups."}
```

Listing 3.2: Handling exceptions in a CKAN extension

Required extensions for prototype

Since the standard installation of CKAN is not directly suitable for publishing geospatial data, we will enable these features through extensions. The extensions we will install as a starting point for our prototype are:

- `ckanext-spatial`: To enable general geospatial features, such as supporting CSW.
- `ckanext-harvest`: To enable harvesting metadata from external repositories, and implement custom harvesters.

- `ckanext-geoview`: To enable rendering of geospatial data.

In addition, we will implement two custom extensions to support harvesting of other data formats. The installation of the three mentioned extensions, and the implementation of our own extensions are described in the next chapter.

Chapter 4

Implementation of GIS prototype

In this chapter we describe how we implement our GIS prototype. We will set up the default CKAN installation, and show that it does not fulfill all of our use cases. Hence, we will enable spatial capabilities by installing various existing extensions. These extensions support many of our use cases out of the box, but not all of them. The missing features will be implemented in our own extensions. Before we implement these, we will experiment with CKANs extension mechanism to figure out how it works. Finally we will present the implementation of our custom extensions.

4.1 Development environment and methodology

The setup of CKAN and the development of its extensions were done using an agile approach. We focused on producing working software, and adapting the code to changing requirements. The development were done in a iterative and exploratory way. We focused on testing several different solutions quickly and choosing the best one, instead of extensively researching literature for the optimal solution first. Thus, we get practical experience with the solutions, and are more likely to find its advantages and disadvantages. This follows from concepts in *lean thinking* [82]. Implementations are done in a virtual machine with Ubuntu 14.04 that were set up using *Vagrant* [65]. The extensions are developed in the editor *Vim* [66].

4.2 CKAN default installation

We start by setting up and configuring the default installation of CKAN. This installation of CKAN is usable as a general Content Management System (CMS) [18]. We will later enable geospatial capabilities for it to be suited for use as a GIS.

4.2.1 Installation procedure

CKAN can be installed in several ways. For development of CKAN, the CKAN developers recommend installing CKAN from source [11]. This can be done by cloning the source code from the git repository [13], and compiling it. For our use cases, we do not have to modify the core source code of CKAN itself, but write extensions. Therefore, we do not have to install from the source code, but can install the CKAN base from a pre built package with a Linux package manager. In August 2017, version 2.7.0 of CKAN was the newest, which is the one we will install. The installation steps are summarised in [28].

To manage CKAN and its dependencies separately from our daily use computer, we will set up a virtual machine and run the installation inside it. We create a virtual machine with Ubuntu 14.04 using Vagrant. Vagrant will automatically download and set up the operation system for us. We then install CKANs dependencies, some of which include:

- Apache2: web server to serve web pages to clients
- NGINX: used as reverse proxy (load balancing, caching common requests)
- Redis: in memory database used as message broker for harvesters
- Pylons: Python web framework (is in deprecation process)
- Flask: Python web framework (to replace Pylons)
- nose: Python testing framework
- SQLAlchemy: object relational mapping
- PostgreSQL: database
- Solr: search platform

Finally, we download the CKAN debian package that contains the CKAN software itself, and install it with the Ubuntu package manager.

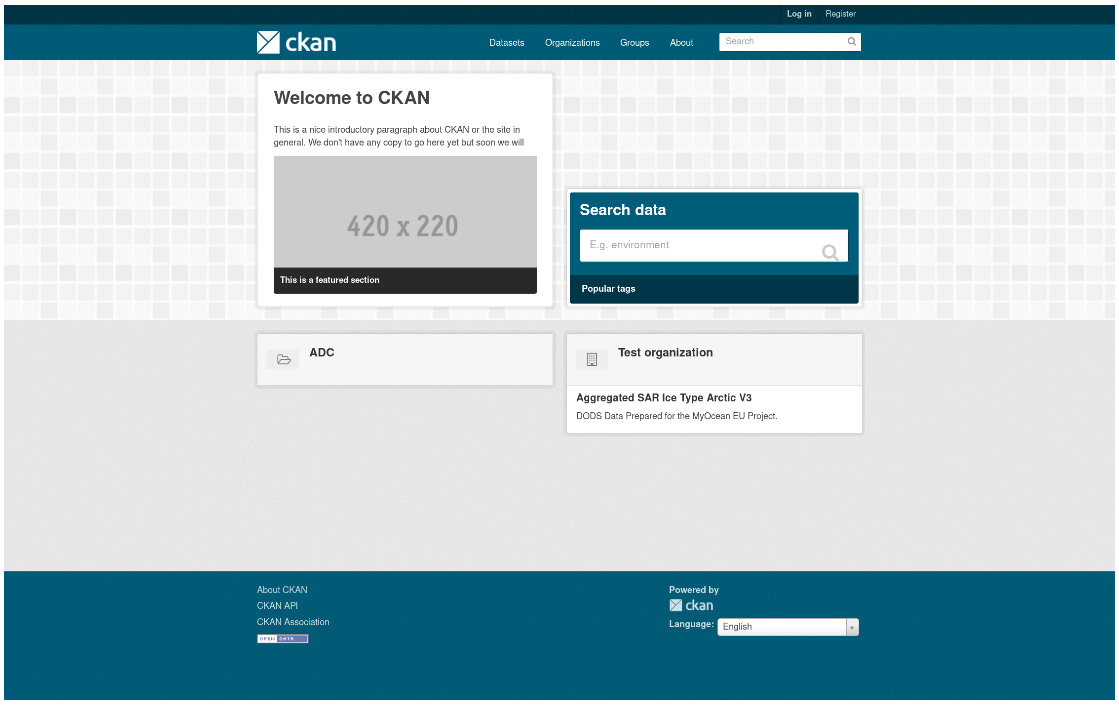
4.2.2 Base configuration

After installing CKAN, we have to make some configurations for the setup to work properly. We create database tables where CKAN will keep its persistent data. We also need separate Linux users on the virtual machine with special permissions to read and write to the database tables. We create two users, one user to run CKAN, and another to access the PostgreSQL database. If the CKAN user is being compromised in a malicious attack, the contents in the database is not accessible by the same user.

For the search functionality to work properly, we also have to configure Apache Solr [53]. Apache Solr is a platform that handles search operations. Features include for example fast, full-text search capabilities and it is flexible so it can be configured for several different use cases [52]. We will provide Solr with a customized Solr schema from the CKAN source code that it will use to be able to search within geospatial data [9]. Both PostgreSQL and Solr is also run on the virtual machine mentioned earlier.

Figure 4.1 shows the front page of the finished base installation. The base CKAN installation includes typical features of content management systems:

- Register user accounts and join organizational groups.
- List and publish simple content like tabular data, text documents and images.
- Search within published content.
- Programmatically access CKAN through the Action API.



Welcome to CKAN

This is a nice introductory paragraph about CKAN or the site in general. We don't have any copy to go here yet but soon we will

Aggregated SAR Ice Type Arctic V3

DODS Data Prepared for the MyOcean EU Project.

Figure 4.1: Front page of CKAN base installation

These features are a good start for our GIS prototype, but we have to extend the base installation to be able to utilise geospatial features. For example the current installation does not support harvesting any metadata from external data servers, a feature required in our prototype.

4.3 Enabling geospatial capabilities

Three extensions exist which provide some of the geospatial functionality we need. `ckanext-spatial` [17], `ckanext-harvest` [16] and `ckanext-geoview` [15] are detailed in the next sections. These extensions are installed using *pip* [41], a tool for installing Python packages, and then enabled in the global CKAN configuration file (Listing A.2). Some of these extensions have various extra dependencies, that need to be installed using methods specific to the operation system being used. For example the harvesting extension depends on a Redis in-memory database that has to be installed manually.

4.3.1 Spatial extension

We enable geospatial capabilities by installing the extension `ckanext-spatial`. The extension adds a spatial field to the CKAN database schema [17]. This allows us to perform spatial searches within the CKAN database, an important operation for users to find the data they need. For example users can search for datasets originating from a specified geographic area. Storing geographic objects in the database is enabled by *PostGIS* [68], an extension to PostgreSQL.

Support for CSW is included with the spatial extension using *pycsw* [86], a python implementation of OGC CSW [86]. CSW allows our prototype to import resources from other CSW servers using a CSW harvester, as well as exposing the resources to external systems and users. Two other harvesters are included in addition to the CSW harvester, one for harvesting from *Web Accessible Folders* [1] containing metadata documents, and one for importing single metadata documents from an URL in the ISO-19139 metadata standard. Validators are implemented in these harvesters. They verify that the format of imported metadata records are correct.

These three extensions make use of a harvester interface that is provided by `ckanext-harvest`. Therefore, to be able to use the harvesters and develop custom ones, we will install the harvesting extension.

4.3.2 Harvesting extension

We need a mechanism to be able to fetch metadata from external locations. It will allow our users to browse datasets that are not directly hosted at our prototype. If the original datasets are changed in the external location, the update will automatically be reflected across the locations where it has been imported. Therefore the user does not have to verify that the dataset is up to date across all the different hosting locations.

The spatial extension provides harvesters for a few standards, but we want to be able to develop harvesters for TDS and OAI-PMH as well. To develop such harvesters, the *general* harvesting extension `ckanext-harvest` have to be installed. This extension can not do any harvesting itself, but it provides some of the components required to do it. For example it provides an interface that will connect harvesters to the CKAN core using the *publish-subscribe pattern.*

Publish-subscribe pattern

The harvesting mechanism use the publish-subscribe pattern to manage communication between several harvesters and the harvesting extension. In the publish-subscribe pattern [47], a publisher publishes messages to *topics* on a queue. The harvesting extension can use Redis [51] or RabbitMQ [50] for the messaging queues. Harvesters interested in topics can subscribe to them, and will be notified when a new message has been published. The subscribers can then consume the message. The pattern is shown in Figure 4.2.

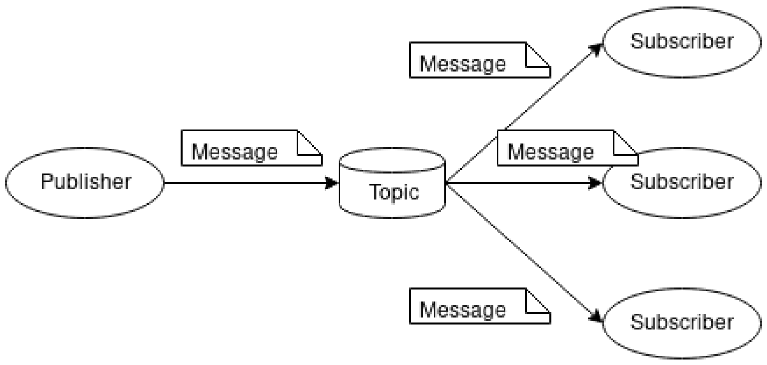
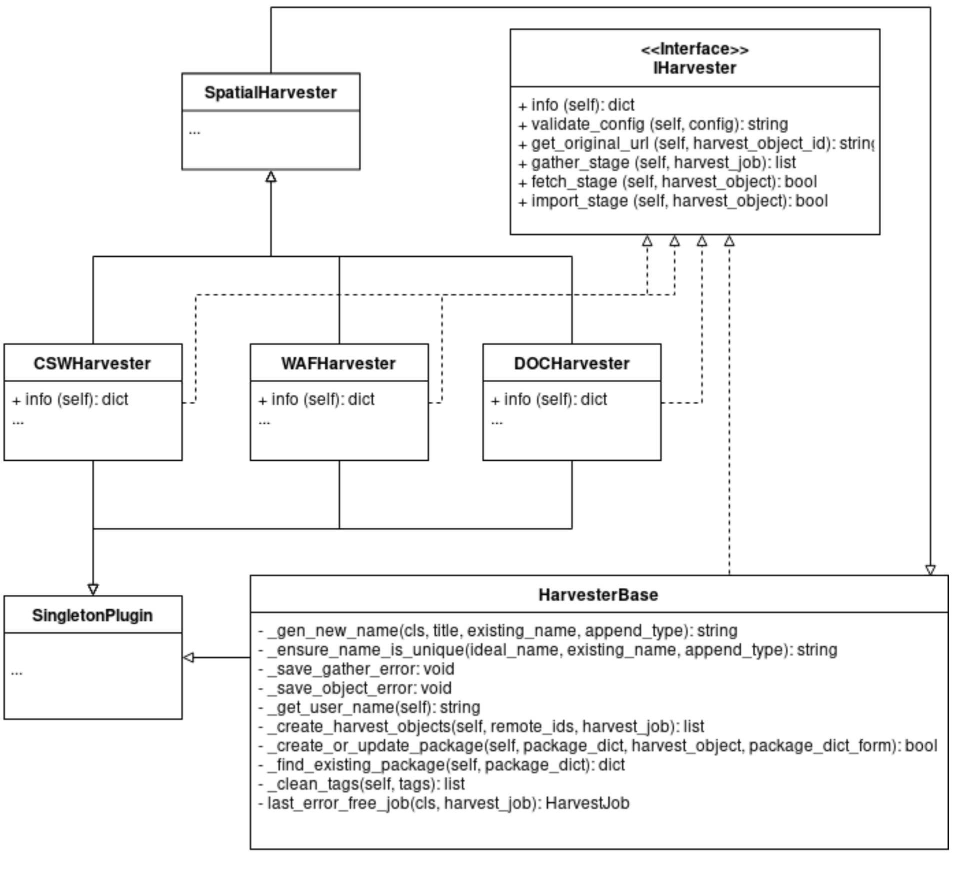


Figure 4.2: Publish-subscribe pattern

We use Redis as the message broker to store messages for the different topics, and to notify the subscribers when new messages have been published. The harvester extension publishes a message to the Redis queue when a new harvest source has been added. The advantage to this pattern is that an arbitrary amount of harvesters can subscribe to the Redis queues, and be notified if any sources with their data type has been added, and respond accordingly. Users can add and remove harvesters (subscribers) dynamically. Without the Redis queue, the harvesting extension would have to be aware of all the specific harvesters available. Because the publishers and subscribers do not know about the existence each other, the coupling is kept low.

Harvester interface

The harvesting extension provides an interface, `IHarvester`, that each harvester must implement to be notified when new harvest sources are added. An overview of the interface and its relations to the harvesters provided by the spatial extension is shown in Figure 4.3. The complete interface is shown in Listing A.1. The harvesters inherit from `SpatialHarvester`, a class that provide some common methods for harvesting spatial data. The `SpatialHarvester` inherit from `HarvesterBase`, that provide some general helper methods for harvesters. Last, the harvesters inherit from `SingletonPlugin`, a base class for extensions where a singleton [84] instance of the class is created when the extension is loaded.



SpatialHarvester
...
<<Interface>>
IHarvester
+ info (self): dict
+ validate_config (self, config): string
+ get_original_url (self, harvest_object_id): string
+ gather_stage (self, harvest_job): list
+ fetch_stage (self, harvest_object): bool
+ import_stage (self, harvest_object): bool
CSWHarvester
+ info (self): dict
...
WAFHarvester
+ info (self): dict
...
DOCHarvester
+ info (self): dict
...
SingletonPlugin
...
HarvesterBase
- _gen_new_name(cls, title, existing_name, append_type): string
- _ensure_name_is_unique(ideal_name, existing_name, append_type): string
- _save_gather_error: void
- _save_object_error: void
- _get_user_name(self): string
- _create_harvest_objects(self, remote_ids, harvest_job): list
- _create_or_update_package(self, package_dict, harvest_object, package_dict_form): bool
- _find_existing_package(self, package_dict): dict
- _clean_tags(self, tags): list
- last_error_free_job(cls, harvest_job): HarvestJob

Figure 4.3: General harvester classes and interfaces

Developers can choose between several strategies when implementing a harvester. Independent of the chosen strategy, all extensions must explicitly or implicitly inherit from `SingletonPlugin`. The most straightforward way to create a harvester is to implement `IHarvester` and develop the harvester logic oneself. Another solution is to extend `HarvesterBase`, implicitly implementing `IHarvester`. Using this method, we can access the helper methods in `HarvesterBase`. A third option is used by the included harvesters in the spatial extensions. They extend `SpatialHarvester`, and therefore access both spatial helper methods and methods from `HarvesterBase`.

Harvesting stages

The mandatory methods we have to implement are `info`, `gather_stage`, `fetch_stage` and `import_stage`. The `info` method returns a description of the harvester. The actual harvesting of metadata is implemented in the three other methods. Signaling that a harvest should run is done either by a user manually clicking a "Reharvest" button in the prototype user interface, or by a periodic timer. When this happens, a harvest job is created, and published to a Redis *gather queue*. A worker will consumer the jobs on this queue, and start the first of the three harvesting stages.

- *Gather*: Gather all the resource identifiers for the resources that should be imported. For a CSW server this stage would query the `GetRecords` operation. The method is responsible for publishing the resource identifiers to the Redis fetch queue, signaling that new resources should be fetched.
- *Fetch*: The fetch method is run for every resource identifier that is published to the Redis fetch queue. Using the resource identifier, the fetch method fetches the actual resource metadata. For a CSW server this stage would query the `GetRecordById` operation. The fetch stage will return true if the harvest ran successfully, and false otherwise. If true is returned, the harvester extension will start the import stage.
- *Import*: The import stage is responsible for transforming the fetched resources to CKAN fields, and store them in the CKAN database. Steps include parsing the data, validating it and creating the CKAN dataset to be displayed to the users.

The gather and fetch stages are run in backgrounds jobs, constantly waiting for new data to harvest. These processes are typically started from a command line interface. In a production environment these processes are managed by for example *Supervisor* [58], a process control and monitoring system. Supervisor can then restart the processes if they stop.

The import stage should be run periodically, removing datasets that were fetched in the fetch stage with errors. If the fetch stage is run successfully, the import stage is run automatically. Therefore, the separate, periodic execution of the import stage can be run for example once every day, with the intention to clean up harvest jobs that returned with an error.

New harvest source flow

Figure 4.4 shows a simplified sequence diagram of how a new harvest source is created in our prototype with the harvesting extension. When a user opens the user interface to add a new source to harvest, the CKAN core queries the harvesting extension for a list of supported harvesters. The user then enters the necessary information about the harvester, such as the source URL. When the data source is saved, the CKAN core calls the harvesting extension to create it. The harvesting extension simply proxies the call back to the *create_package* action in the CKAN Action API. This action will create a dataset *package*; how datasets and harvest sources are represented in CKAN.

Finally, the harvest source is persisted in the PostgreSQL database. At this stage a harvest source is simply created, but no metadata is harvested yet. For this we need to implement a custom harvester, or enable one of the existing ones. The implementation of such harvesters are explained in Section 4.5 and Section 4.6.

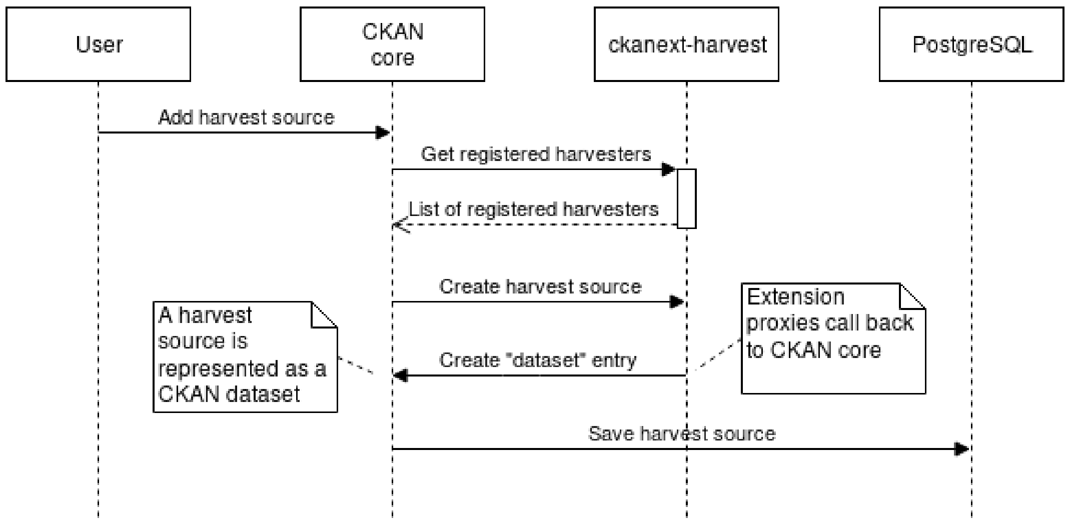

User
CKAN core
ckanext-harvest
PostgreSQL
Add harvest source
Get registered harvesters
List of registered harvesters
Create harvest source
Extension proxies call back to CKAN core
A harvest source is represented as a CKAN dataset
Create "dataset" entry
Save harvest source

Figure 4.4: Adding a new source to harvest

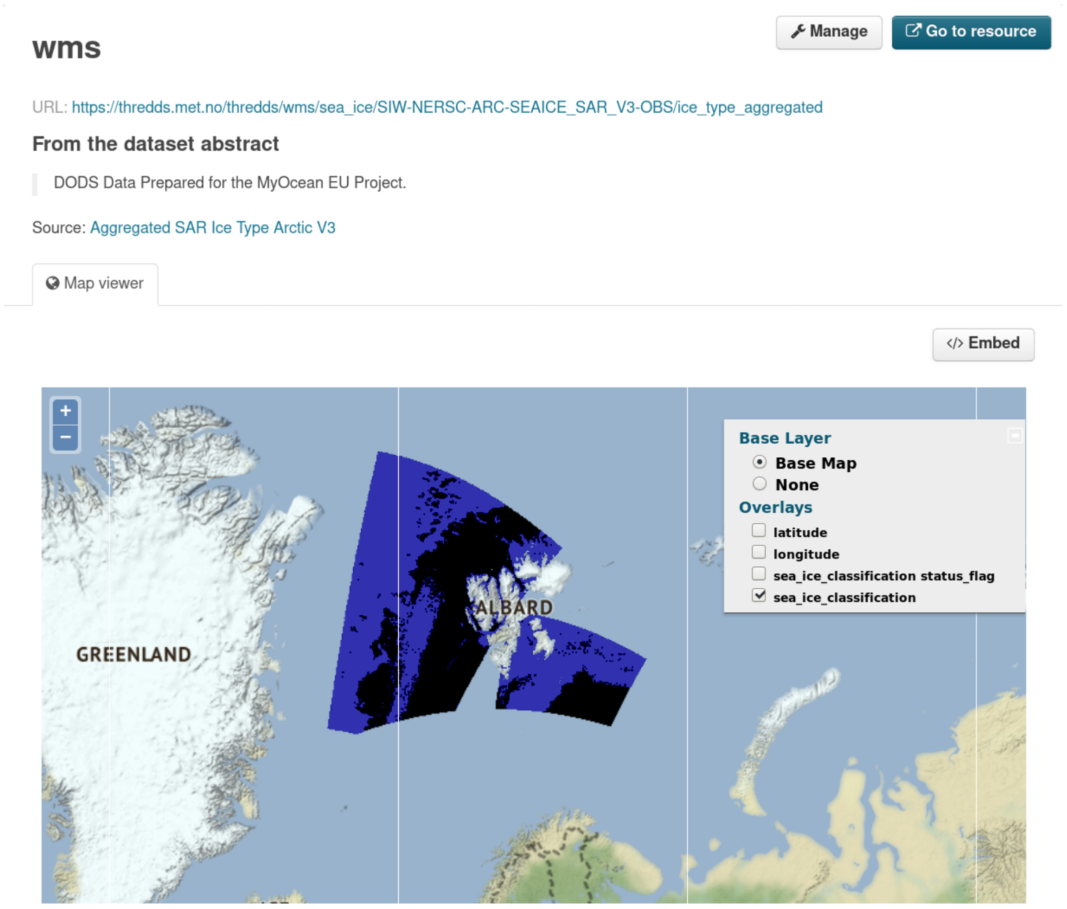
4.3.3 Geographic view extension

Rendering of geospatial resources is not included in the spatial extension. These views have to be provided by a separate extension, `ckanext-geoview` [15]. The viewers in this extension are implemented using *OpenLayers* [40] or *Leaflet* [29], and provides rendering support for various formats and protocols. These are listed below.

- Web Map Service (WMS)
- Web Feature Service (WFS)
- Web Map Tile Service (WMTS)

The extension can also render data in GeoJSON, Geography Markup Language (GML) and Keyhole Markup Language (KML). We can implement the interface `IResourceView` [43] to create viewers that render new formats.

Figure 4.5 shows how the geographic view extension renders a WMS service. Several base maps (base layers) can be added, and chosen in the dialog to the right in the view. In this dialog we can also select and deselect the various layers that the service provide.


wms

URL: https://thredds.met.no/thredds/wms/sea_ice/SIW-NERSC-ARC-SEAICE_SAR_V3-OBS/ice_type_aggregated

From the dataset abstract

DODS Data Prepared for the MyOcean EU Project.

Source: Aggregated SAR Ice Type Arctic V3

Base Layer
[● Base Map]
○ None

Overlays
☐ latitude
☐ longitude
☐ sea_ice_classification status_flag
☑ sea_ice_classification

Figure 4.5: Rendering a WMS service with the geographical view extension

4.4 Experimenting with custom extensions

Before we start developing harvesters, it is useful to experiment with the extension mechanism beforehand. We do this to learn how extensions work in practise, and to gain insight in how to effectively develop the actual harvesters. Without experimenting with extensions first, we might start developing a harvester using bad practises, and might end up having to do a complete rewrite of the harvester at a later stage. Using the experimenting approach, we can make these early mistakes without any loss.

4.4.1 Generate and enable extension

In the base installation of CKAN, a configuration for a *Python virtual environment* [67] is included. Such a virtual environment is a self-contained directory with a specific Python version installed. In this environment specific versions of Python libraries are also installed. Using the self-contained environment, CKAN and its dependencies are kept separate from other installations.

Inside the CKAN virtual environment, we can run the command `paster create` to generate an empty extension, with all required directories and files.

```
ckanext-our-test-extension/
├── ckanext/
│   ├── __init__.py
│   └── our-test-extension/
│       ├── __init__.py
│       └── plugin.py
├── ckanext_our-test-extension.egg-info/
└── setup.py
```

The relevant source for our extension will reside in the `our-test-extension` directory. In that directory we can create a simple `plugin.py`, as seen in Listing 4.1. For now the file contains a single print statement, that we will use to verify that our extension is enabled and working. We will see in Section 3.5.2 how we can access the CKAN core and customise its features.

```python
import ckan.plugins as plugins

class TestExtension(plugins.SingletonPlugin):
    print('This is our test extension!')
```

Listing 4.1: Plugin class with single print statement

We add the path and class name of our plugin in `setup.py` on lines 17-20 as seen in Listing 4.2. This will define our plugin class as the entry point for our extension. When we enable the extension in the global CKAN configuration file, it will be referenced with the name *test*, pointing to the `TestPlugin` class in the `ckanext` namespace, as defined on line 14. If our extension requires specific dependencies, these are defined in the `install_requires` list on line 15. This is kept empty for now.

```
 1  from setuptools import setup, find_packages
 2  setup(
 3      name='''ckanext-test''',
 4      version='0.0.1',
 5      description='''Test extension''',
 6      long_description='',
 7      url='https://github.com/andrmos/ckanext-test',
 8      author='''Test user''',
 9      author_email='''user@test.com''',
10      license='AGPL',
11      classifiers=[],
12      keywords='''CKAN test''',
13      packages=find_packages(exclude=['contrib', 'docs', 'tests*']),
14      namespace_packages=['ckanext'],
15      install_requires=[],
16      include_package_data=True,
17      entry_points='''
18          [ckan.plugins]
19          test=ckanext.test.plugin:TestPlugin
20      '''
21  )
```

Listing 4.2: Extension configuration

We use the `setup.py` file for installing our test extension. This is done by running the command `python setup.py develop`. Last, we enable our extension by adding it in the global CKAN configuration file, `production.ini`. When we reload CKAN, the plugin prints *This is our test extension!* to the CKAN logs.

4.4.2 Testing the extension

To write and run tests for our extension, we create a `test.ini` configuration file in the `ckanext-our-test-extension/` directory. This file will contain configuration options that will be used when running tests. We will also create a new directory, `ckanext-our-test-extension/tests/`, where our tests will reside. In this directory we can create a python file `test_our-test-extension.py`. This file will contain all our test methods.

Nosetests will run all methods with *"test"* as part of the method name. We add a unit test named using the unit test naming convention. It contains a simple assertion to verify that the tests are running correctly. The test is shown in Listing 4.3.

Executing the command
`nosetests -v --ckan --with-pylons=test.ini ckanext/our-test-extension/tests`
will run the test, and give the output in Listing 4.4. In a real extension, we would add several unit tests. For example a real unit test for a harvesting extension could create a harvest job object from a harvest source, and assert that both of their identifiers are the same. We will now start developing the extensions needed to achieve our goals from Section 1.3.

```python
def test_one_plus_one_should_equal_two():
    a = 1
    b = 1
    assert a + b == 2
```

Listing 4.3: Simple unit test

```
test_our-test-extension.test_one_plus_one_should_equal_two ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Listing 4.4: Unit test output

4.5 Customise existing OAI-PMH harvester

A harvester for OAI-PMH exists [91] that we will use as a starting point for our harvester, and modify to our needs. This harvester uses *pyoai* [48], a Python implementation of a OAI-PMH client and server. An example of how one could use pyoai in our extension to fetch resource identifiers from an OAI-PMH repository is shown in Listing 4.5. We import the client, and provide it with the repository URL and a `MetadataRegistry`. The `MetadataRegistry` contains `MetadataReaders` used for parsing the XML responses of pyoai. Last, we call one of the OAI-PMH operations. More options can be provided to the client to customise its behaviour.

```python
import oaipmh.client
client = oaipmh.client.Client(
    'www.example.com',
    metadata_registry
)

for header in client.listIdentifiers(metadataPrefix='dif'):
    resource_id = header.identifier()
    # Do something with resource_id...
```

Listing 4.5: OAI-PMH Client usage example

The OAI-PMH harvester is missing some features we need. For example it only provides support for metadata in Dublin Core format [22]. We need support for DIF as well, since it is better suited for geographic data. Additionally, it does not handle configurations entered by users well, and crashes if no configuration is provided.

4.5.1 DIF metadata reader

The first step is to implement support for reading metadata in DIF. The `MetadataReader` class in `metadata.py` contains the business logic for reading metadata from an OAI-PMH repository. It uses the *lxml* toolkit [30] to parse metadata. lxml adds Python bindings to the C libraries *libxml2* [70] and *libxslt* [71], that provide efficient XML parsing. The `MetadataReader` locates specific fields in the metadata using *XPath* evaluations, and sets its value to fields that will be displayed in the web interface in our prototype. Listing 4.6 shows how the metadata reader maps Dublin Core metadata to CKAN fields. On line 3, the `title` field is set to the `text` content of the child element of `oai_dc:dc`: `dc:title`. A simplified example of how the XML might look is shown in Listing 4.7.

```python
1  oai_dc_reader = MetadataReader(
2  fields={
3      'title':            ('textList', 'oai_dc:dc/dc:title/text()'),
4      'creator':          ('textList', 'oai_dc:dc/dc:creator/text()'),
5      'subject':          ('textList', 'oai_dc:dc/dc:subject/text()'),
6      'description':      ('textList', 'oai_dc:dc/dc:description/text()'),
7      'publisher':        ('textList', 'oai_dc:dc/dc:publisher/text()'),
8      'maintainer_email': ('textList', 'oai_dc:dc/oai:maintainer_email/text↩
           ()'),
9      'contributor':      ('textList', 'oai_dc:dc/dc:contributor/text()'),
10     'date':             ('textList', 'oai_dc:dc/dc:date/text()'),
11     'type':             ('textList', 'oai_dc:dc/dc:type/text()'),
12     'format':           ('textList', 'oai_dc:dc/dc:format/text()'),
13     'identifier':       ('textList', 'oai_dc:dc/dc:identifier/text()'),
14     'source':           ('textList', 'oai_dc:dc/dc:source/text()'),
    'language':         ('textList', 'oai_dc:dc/dc:language/text()'),
    'relation':         ('textList', 'oai_dc:dc/dc:relation/text()'),
    'coverage':         ('textList', 'oai_dc:dc/dc:coverage/text()'),
    'rights':           ('textList', 'oai_dc:dc/dc:rights/text()')
},
namespaces={
    'oai_dc': 'http://www.openarchives.org/OAI/2.0/oai_dc/',
    'oai': 'http://www.openarchives.org/OAI/2.0/',
    'dc': 'http://purl.org/dc/elements/1.1/'
}
)
```

Listing 4.6: Metadata reader for Dublin Core

```
<oai_dc:dc>
    <dc:title>AROME METCOOP 0.5 km</dc:title>
</oai_dc:dc>
```

Listing 4.7: Dublin Core metadata example

Additional metadata readers can be implemented in the script `metadata.py`. These readers have to be registered with the pyoai client that the extension uses. The registration of those readers are done in `_create_metadata_registry()`, shown in Listing 4.8. When we have created our own metadata reader for DIF, it is registered as shown on line 6. The *'oai_dc'* and *'dif'* strings are keys used to identify the metadata readers. The OAI-PMH client is given the registry upon initialisation, and chooses metadata reader based on resource formats available (*ListMetadataFormats* OAI-PMH operation).

```
1  from metadata import oai_dc_reader, dif_reader
2
3  def _create_metadata_registry(self):
4      registry = MetadataRegistry()
5      registry.registerReader('oai_dc', oai_dc_reader)
6      registry.registerReader('dif', dif_reader)
7      return registry
```

Listing 4.8: Registration of metadata readers

We create our own metadata reader for DIF in the script `metadata.py`. Part of our reader is shown in Listing 4.9. The complete reader is not shown due to the amount of fields in DIF.

```
dif_reader = MetadataReader(
    fields={
        # Basic info
        "Entry_ID": _eval_builder('textList', ['Entry_ID', 'text()']),
        "Entry_Title": _eval_builder('textList', ['Entry_Title', 'text()']),

        # Dataset citation
        "Data_Set_Citation/Dataset_Creator": _eval_builder('textList', ['Dataset_Creator', 'text()']),
        "Data_Set_Citation/Dataset_Title": _eval_builder('textList', ['Dataset_Title', 'text()']),
        "Data_Set_Citation/Dataset_Release_Date": _eval_builder('textList', ['Dataset_Release_Date', '↩
            text()']),
        "Data_Set_Citation/Dataset_Release_Place": _eval_builder('textList', ['Dataset_Release_Place', '↩
            text()']),
        "Data_Set_Citation/Dataset_Publisher": _eval_builder('textList', ['Dataset_Publisher', 'text()'↩
            ]),
        "Data_Set_Citation/Version": _eval_builder('textList', ['Version', 'text()']),

        #...
    },
    namespaces={
        'dif': 'https://gcmd.nasa.gov/Aboutus/xml/dif/'
    }
)
```

Listing 4.9: Part of DIF metadata reader

The DIF metadata from data sources we harvest does not contain any *XML namespace* [63]. Therefore we can not parse the DIF metadata the same way it is done with Dublin Core. A query looking for the text of an element with the name `title` anywhere within the XML looks like this: `//*[name()='title']/text()`. The problem of handling elements with identical name is described later in Section 4.7. We have implemented a separate method that creates such queries, as they tend to be long if written manually. The logic to construct such XPath queries are shown in Listing 4.10.

```python
# Builds an XPath based on a list of elements, ex: //*[name()='title']/text()
# For XML parsing without namespaces
def _xpath_bulder(elms):
    path = ""
    for i, elm in enumerate(elms):
        if i == len(elms) - 1:
            path += "/"
            path += elm
        else:
            path += "//*[name()='"
            path += elm
            path += "']"
    return path


def _eval_builder(field_type, elms):
    return (field_type, _xpath_bulder(elms))
```

Listing 4.10: Evaluation and XPath builder

4.5.2 Harvester overview

The relationship between the OAI-PMH harvester and the other harvesters are shown in the class diagram in Figure 4.6. We have omitted methods and attributes that are irrelevant to the implementation of the OAI-PMH harvester. The relationships in the diagram are complex as the `CSWHarvester`, `WAFHarvester` and `DOCHarvester` from the spatial extension uses multiple inheritance. A simpler strategy is chosen for the OAI-PMH harvester.

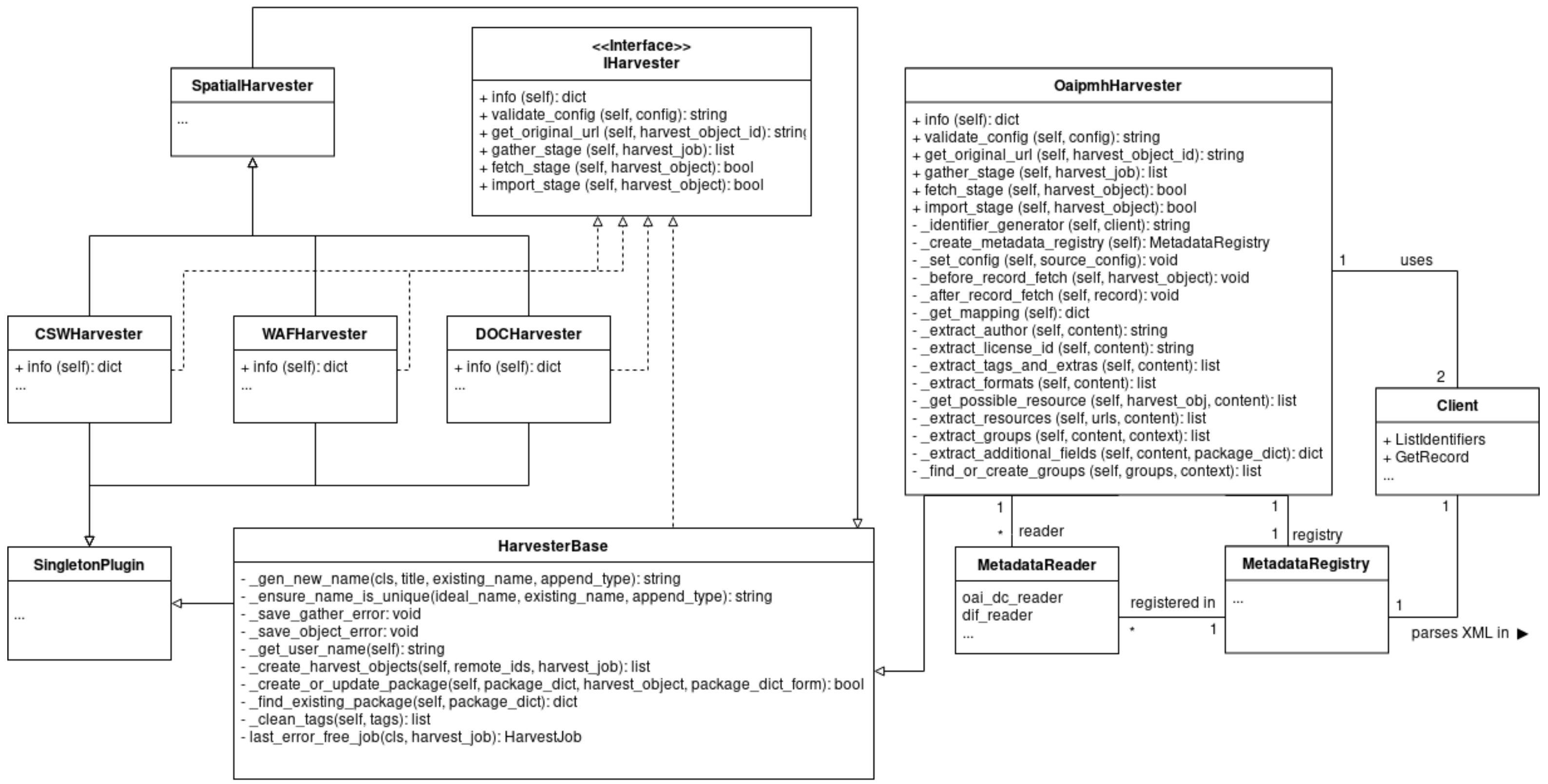

SpatialHarvester
...
<<Interface>>
IHarvester
+ info (self): dict
+ validate_config (self, config): string
+ get_original_url (self, harvest_object_id): string
+ gather_stage (self, harvest_job): list
+ fetch_stage (self, harvest_object): bool
+ import_stage (self, harvest_object): bool
OaipmhHarvester
+ info (self): dict
+ validate_config (self, config): string
+ get_original_url (self, harvest_object_id): string
+ gather_stage (self, harvest_job): list
+ fetch_stage (self, harvest_object): bool
+ import_stage (self, harvest_object): bool
- _identifier_generator (self, client): string
- _create_metadata_registry (self): MetadataRegistry
- _set_config (self, source_config): void
- _before_record_fetch (self, harvest_object): void
- _after_record_fetch (self, record): void
- _get_mapping (self): dict
- _extract_author (self, content): string
- _extract_license_id (self, content): string
- _extract_tags_and_extras (self, content): list
- _extract_formats (self, content): list
- _get_possible_resource (self, harvest_obj, content): list
- _extract_resources (self, urls, content): list
- _extract_groups (self, content, context): list
- _extract_additional_fields (self, content, package_dict): dict
- _find_or_create_groups (self, groups, context): list
1
uses
2
Client
+ ListIdentifiers
+ GetRecord
...
CSWHarvester
+ info (self): dict
...
WAFHarvester
+ info (self): dict
...
DOCHarvester
+ info (self): dict
...
1
*
reader
1
1
registry
1
MetadataReader
oai_dc_reader
dif_reader
...
registered in
*
1
MetadataRegistry
...
1
parses XML in ▶
SingletonPlugin
...
HarvesterBase
- _gen_new_name(cls, title, existing_name, append_type): string
- _ensure_name_is_unique(ideal_name, existing_name, append_type): string
- _save_gather_error: void
- _save_object_error: void
- _get_user_name(self): string
- _create_harvest_objects(self, remote_ids, harvest_job): list
- _create_or_update_package(self, package_dict, harvest_object, package_dict_form): bool
- _find_existing_package(self, package_dict): dict
- _clean_tags(self, tags): list
- last_error_free_job(cls, harvest_job): HarvestJob

Figure 4.6: OAI-PMH harvester class diagram

`OaipmhHarvester` inherits from `HarvesterBase`, a generic base class that implements the `IHarvester` interface. The `HarvesterBase` provide helper methods that are useful for harvesters. For example the `_create_or_update_package()` method creates a new dataset package if it does not already exist, or it updates an existing one. Such methods are helpful since the logic is the same for most harvesters, and they do not end up cluttering the logic of the specific harvesters.

The logic for making the OAI-PMH requests reside in `Client`, a class provided by *pyoai*. This class contains implementations for the `ListIdentifiers` and `GetRecord` operations, among others, in OAI-PMH. Recall the code in the example in Listing 4.5.

The `MetadataRegistry` and `MetadataReader` classes contain the logic for parsing XML data and transforming them to Python objects. The registry contain several metadata readers, that are implemented in the `MetadataReader` class. The registry is used by the `Client` to parse the XML responses it receives. Two instances of `Client` is used in `OaipmhHarvester`: one in `gather_stage` and one in `fetch_stage`.

4.5.3 Data flow

Figure 4.7 presents a sequence diagram of how the OAI-PMH harvester works. It shows its connection to the CKAN core and the general harvester extension from Section 4.3.2. The diagram is simplified, displaying the most relevant interactions within the harvester.

The harvesting procedure is started manually by a user clicking a button in the user interface or by a timer. For example a harvest can be scheduled to start automatically every 6 hours. The harvester extension receives the click event, and creates a `HarvestJob` with name *OAI-PMH*, that is published to the Redis gather queue. Our OAI-PMH harvester is subscribed to the gather queue, and is therefore notified when new `HarvestJobs` are created. When it is notified, the gather stage is started, and calls the OAI-PMH repository with the `ListIdentifiers` operation. The operation returns the resource identifiers to fetch. Note 1 and 2 in Figure 4.7 refers to several other calls that are made in the gather stage, but are omitted to simplify the sequence diagram. For example `MetadataReader` objects are registered to a `MetadataRegistry` that is provided to the pyoai client. The returned resource identifiers are published to the Redis fetch queue.

The fetch stage is started when our harvester is notified of the resource identifiers published to the fetch queue. A `GetRecord` request based on a resource identifier is made, and the resource is returned. An example of how a `GetRecord` request looks is shown below.

```
http://arcticdata.met.no/metamod/oai?verb=GetRecord&metadataPrefix=dif
 &identifier=urn:x-wmo:md:no.met.arcticdata.test3::ADC_met-arome-0p5km
```

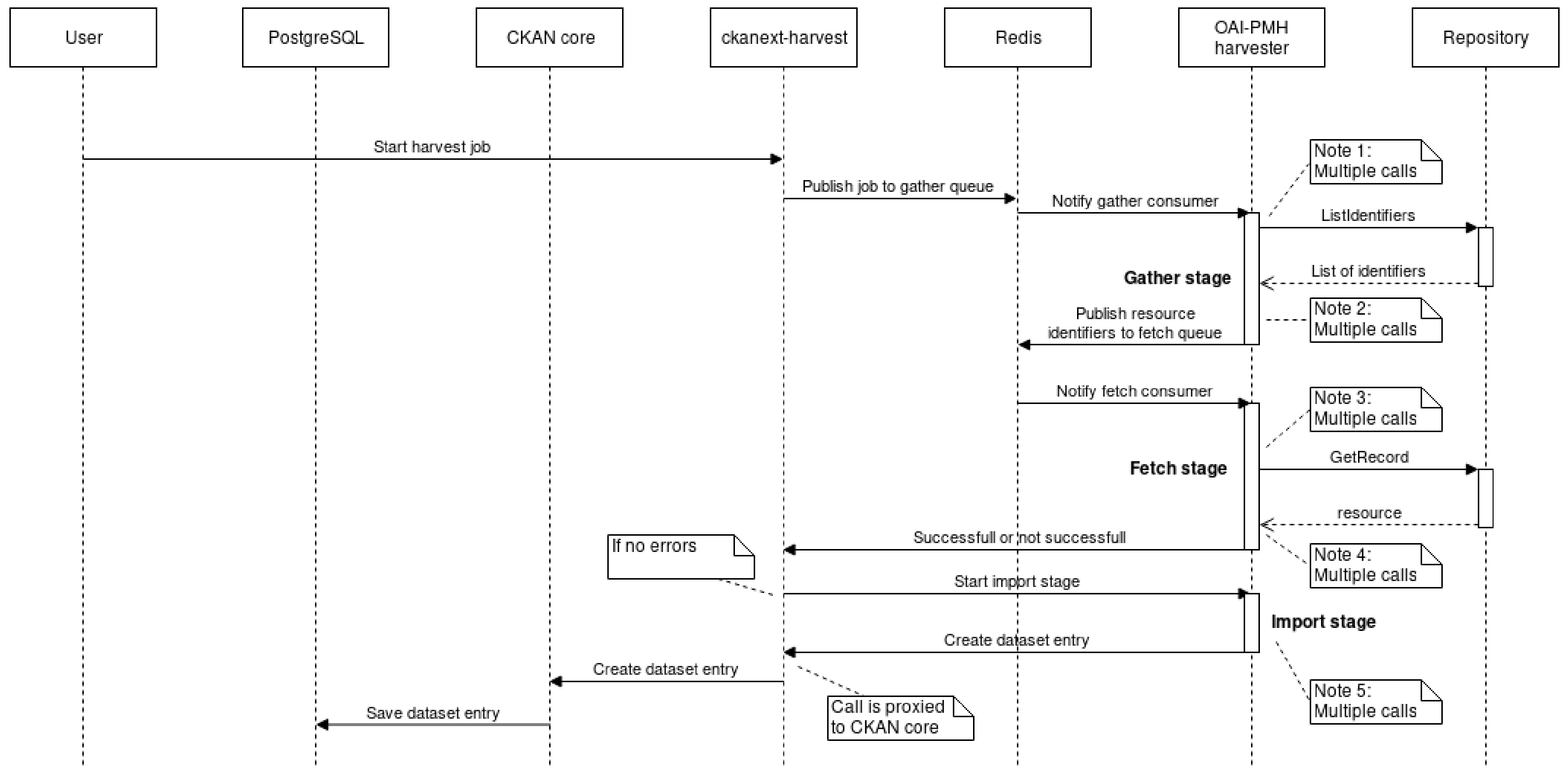


Figure 4.7: Sequence diagram of OAI-PMH harvester

A DIF metadata format is requested for a resource with identifier `urn:x-wmo:md:no.met.arcticdata.test3::ADC_met-arome-0p5km`. Several other calls are also done in the fetch stage, but these are omitted from Figure 4.7 to simplify the diagram.

If no errors are returned by the fetch stage, the import stage is started. This stage transforms the returned resources into appropriate fields for use and display in the prototype. When a resource is transformed, it is saved to the CKAN database, and a call to `package_create` is made. This call is simply proxied to the CKAN core using the plugin toolkit. A CKAN dataset is now created that can be displayed in the web interface.

4.5.4 Various smaller changes

Support several resources in a dataset

We also had to do some smaller miscellaneous changes to the OAI-PMH harvester. For example it only supported adding a single resource to a dataset. If we request the metadata in Dublin Core format, we would only receive a link to a TDS catalogue. This way, the link to the TDS catalogue would be displayed in the prototype. To access the data, the user would have to navigate to the data catalogue first, before being able to access the links to WMS and OPeNDAP.

In DIF, in addition to a link to the TDS catalogue, we also receive direct links to WMS and OPeNDAP services. We would like to store all these resources in a single CKAN dataset. As a result, the user can directly choose how they want to access the data, without navigating to the TDS catalogue first. A simplified implementation of the `_extract_resources()` method is shown in Listing 4.11.

```python
def _extract_resources(self, urls, content):
    if self.md_format == 'dif':
        resources = []
        if urls:
            try:
                resource_formats = self._extract_formats(content)
            except (IndexError, KeyError):
                # Raise error... Omitted from listing.
            for index, url in enumerate(urls):
                resources.append({
                    'name': content['Related_URL/Description'][index],
                    'resource_type': resource_formats[index],
                    'format': resource_formats[index],
                    'url': url
                })
        return resources
```

Listing 4.11: Extracting resources in DIF format

Detecting resource types

We need a way to detect the formats of resources in DIF. When CKAN is provided the type of a resource, it can render the types in different views. For example a WMS service should be of type `wms`. When CKAN receives a WMS service, it knows that it should use the geographic view extension and render the WMS viewer. The detection of resource types is done by checking the content of the access URL. For example if the URL contains *wms*, it is a WMS service, if it contains *dods*, it is a OPeNDAP service, and if it contains *catalog*, it is a TDS catalogue resource. This is a naive implementation and should be improved to be more robust in future work.

4.6 Implementing TDS harvester

The TDS harvester is similar to the OAI-PMH harvester in terms of structure, since they both implement the same harvester interface. Software exist for fetching datasets from a TDS catalogue. Our TDS harvester will be based on this software.

4.6.1 TDS crawler

The TDS harvester is based on `thredds_crawler` [61]. This Python package crawls and parses TDS catalogues recursively using *depth-first search*, and stores the datasets as Python objects. The complete metadata of the datasets in a data catalogue is accessible directly as XML. We can parse the metadata XML with lxml. The crawler supports many options, such as filtering by dataset name with Python regular expressions and ignoring certain datasets based on file type or creation date. We can also customise the number of *workers* (processes) used for crawling a catalogue, to improve performance. A short example of how to crawl a TDS catalogue with `thredds_crawler` is demonstrated in Listing 4.12. The code would fetch all datasets from `‘‘www.example.com’’` created after January 1st 2018, and might print the output similar to line 6 and onwards.

```
1 from thredds_crawler.crawl import Crawl
2 c = Crawl('www.example.com', after=datetime(2018, 1, 1))
3 print(c.datasets[0].name)
4 print(c.datasets[0].services)
5
```

```
SVIM ocean hindcast archive
[{
    'url': 'www.link-to-wms-endpoint.com',
    'name': 'wms',
    'service': 'WMS'
}]
```

Listing 4.12: Example use of the TDS crawler

4.6.2 Harvester overview

Figure 4.8 shows a class diagram of the TDS harvester, giving an overview of its implementation. We have omitted other harvesters and classes to simplify the diagram. The `ThreddsHarvester` inherits from `HarvesterBase`, which implements the `IHarvester` interface as in the OAI-PMH harvester. The `Crawl` class is used by the harvester and is provided by the TDS crawler. `Crawl` is responsible for crawling the TDS catalogues, and return the datasets for use in the harvester. The `LeafDataset` class represents datasets in a TDS catalogue. The private methods in `ThreddsHarvester` are specific to this harvester. These methods are primarily used for modifying the datasets returned by `Crawl` to conform to the CKAN dataset package schema. Refer to Listing A.3 for the complete implementation of `ThreddsHarvester`.

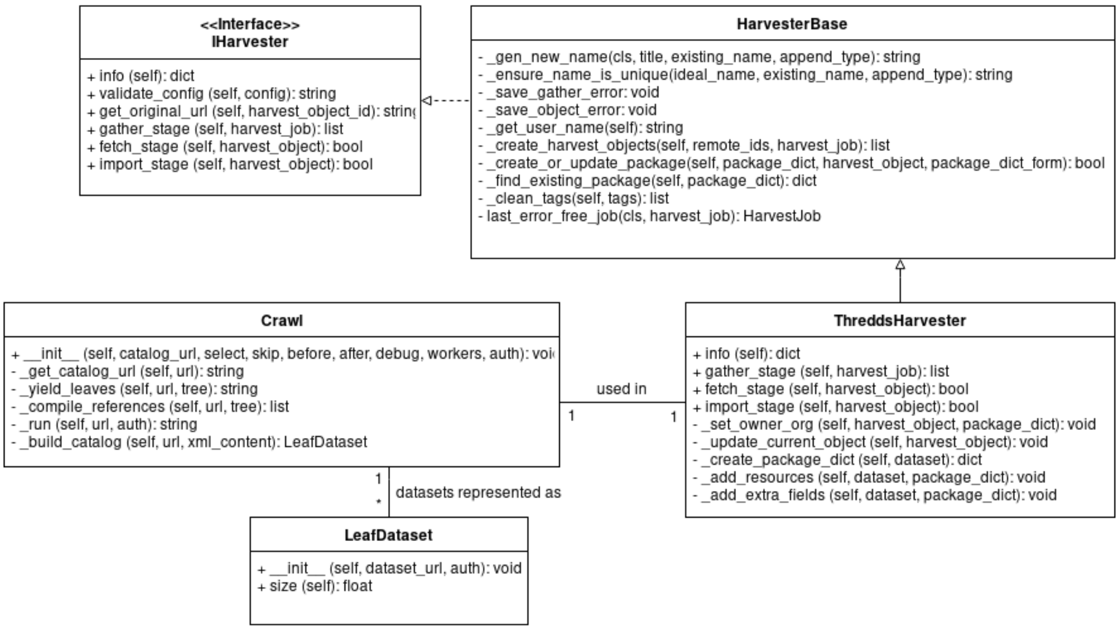

<<Interface>>
IHarvester
+ info (self): dict
+ validate_config (self, config): string
+ get_original_url (self, harvest_object_id): strin
+ gather_stage (self, harvest_job): list
+ fetch_stage (self, harvest_object): bool
+ import_stage (self, harvest_object): bool
HarvesterBase
- _gen_new_name(cls, title, existing_name, append_type): string
- _ensure_name_is_unique(ideal_name, existing_name, append_type): string
- _save_gather_error: void
- _save_object_error: void
- _get_user_name(self): string
- _create_harvest_objects(self, remote_ids, harvest_job): list
- _create_or_update_package(self, package_dict, harvest_object, package_dict_form): bool
- _find_existing_package(self, package_dict): dict
- _clean_tags(self, tags): list
- last_error_free_job(cls, harvest_job): HarvestJob
Crawl
+ __init__ (self, catalog_url, select, skip, before, after, debug, workers, auth): voi
- _get_catalog_url (self, url): string
- _yield_leaves (self, url, tree): string
- _compile_references (self, url, tree): list
- _run (self, url, auth): string
- _build_catalog (self, url, xml_content): LeafDataset
used in
1
1
ThreddsHarvester
+ info (self): dict
+ gather_stage (self, harvest_job): list
+ fetch_stage (self, harvest_object): bool
+ import_stage (self, harvest_object): bool
- _set_owner_org (self, harvest_object, package_dict): void
- _update_current_object (self, harvest_object): void
- _create_package_dict (self, dataset): dict
- _add_resources (self, dataset, package_dict): void
- _add_extra_fields (self, dataset, package_dict): void
1
*
datasets represented as
LeafDataset
+ __init__ (self, dataset_url, auth): void
+ size (self): float

Figure 4.8: TDS harvester class diagram

4.6.3 Data flow

A sequence diagram of harvesting from a TDS catalogue is shown in Figure 4.9. The diagram is simplified to show the most relevant operations. Identical to other harvesters, this sequence diagram features `gather_stage`, `fetch_stage` and `import_stage`. Because crawling a data source for data is done in a single pass (unlike OAI-PMH which requests identifiers first and then fetches the resources afterwards), the implementation of a TDS harvester is simpler.

The harvesting is started equivalent to the OAI-PMH harvester. When the TDS harvester is notified of a new harvest job, the crawling of the data catalogue is started using `thredds_crawler`. The TDS catalogue returns all resources requested. The returned data is added to a Python directory, with fields conforming to the CKAN package schema.

Instead of publishing just the resource identifiers to the fetch queue, the complete resources are published. In this scenario, the gather stage also performs the tasks of the fetch stage. Therefore, the fetch stage simply returns true. If an error happens in the gather stage, resources will not be published to the fetch queue, and the fetch stage will not start.

When the fetch stage returns true, the import stage is started. Since the datasets returned by the TDS catalogue is already transformed to the CKAN schema, the import stage simply calls the `package_create` operation to create the dataset for display in the prototype. Last, the dataset is persisted in the CKAN database.

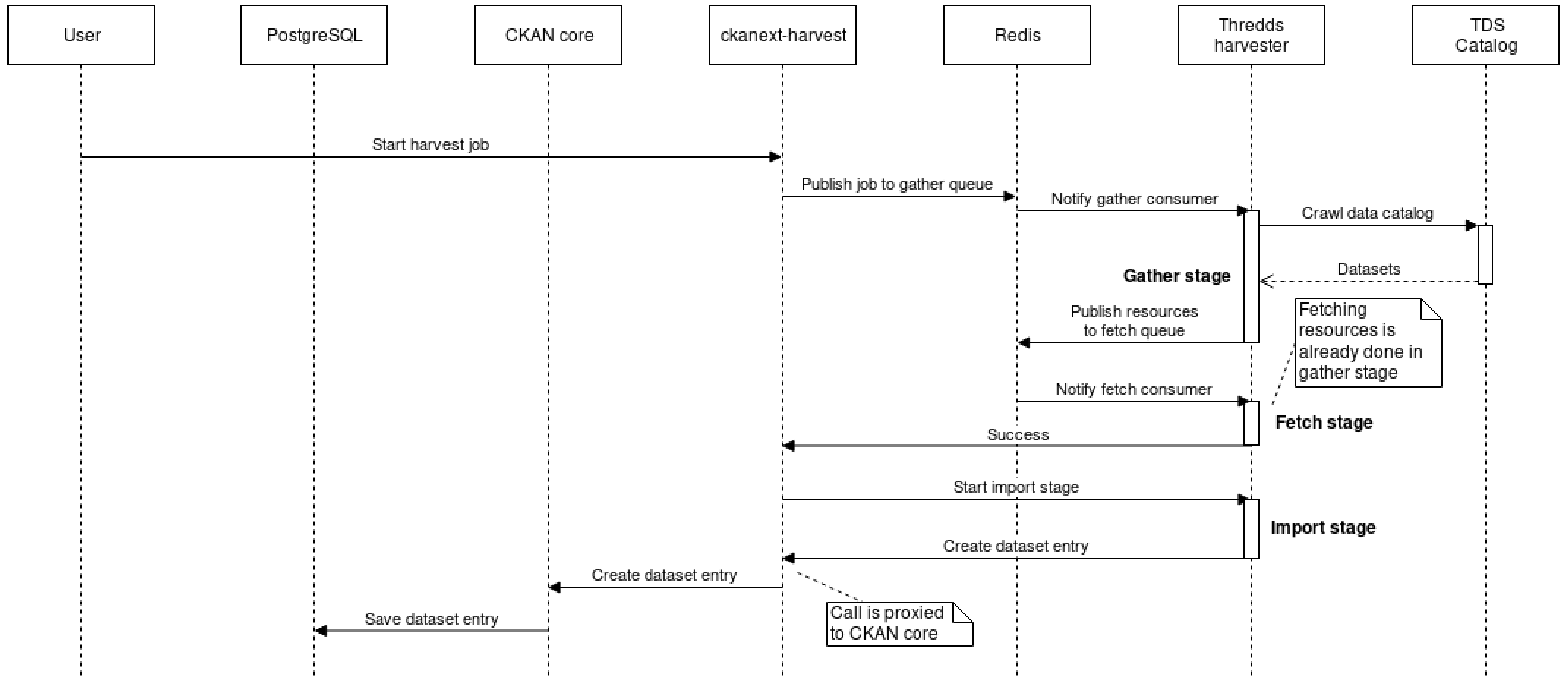


Figure 4.9: TDS harvester sequence diagram

4.7 Implementation challenges and solutions

A few complications took considerable time to figure out how to solve. When the solution was found, they required minor code changes to be fixed. These errors would less likely occur with more experience with CKANs extension mechanism and Python development in general. We will explain some of the challenges we encountered when implementing CKAN extensions, and how they were solved.

1. One of the problems encountered were related to accessing the metadata of datasets returned by `thredds_crawler` version 1.5.3. After crawling a TDS catalogue with the `Crawl` class, we try to access the returned metadata using `dataset.metadata`. This raised an exception:

   ```
   Assertion error:  invalid proxy at 123123123892
   ```

   We did not find a complete explanation of why this exception is raised. The stack trace of the error reveals that it might be related to lxml. A series of Stack Overflow posts [57, 56] and bug reports[31, 62] indicate that the following is likely to have caused the problem.

   The error is raised by lxml, used for parsing XML in `thredds_crawler`. Recall that lxml provides Python bindings to the C libraries *libxml2* and *libxslt*. When lxml parses XML, it represents the parsed XML as C data structures. Additionally, `thredds_crawler` uses multiple threads to crawl data catalogues. `thredds_crawler` returns `lxml.etree.Element` classes, using the C representation of the element. When these elements are accessed outside of the worker process in our harvester extension, the according C representation of the elements cannot be found, and the error is raised.

   The issue was fixed in a new release of `thredds_crawler`, version 1.5.4 [60], released 16th of March 2018. In this version, `thredds_crawler` does not store the `lxml.etree.Element` class directly, but instead deserializes the element to a string with `etree.tostring(metadata)`. We can access the string in our extension without issues, since we no longer implicitly try to access the C representation outside its process.

2. Another problem was also related to parsing XML. Normally XML should contain *namespaces* [63] to be able to distinguish between XML elements with the same name. Responses in Dublin Core format from OAI-PMH contained namespaces and were easy to parse with lxml. Responses from TDS and OAI-PMH in DIF format did not contain any namespaces for the metadata. Therefore it had to be parsed with a different syntax. Instead of simply accessing elements with for example `oai_dc:dc/dc:title/text()`, we have to use for example `//*[name()='title']/text()`. This syntax is longer for deeply nested structures, where you have to define the complete XPath from the root node. The queries are more confusing to read and understand, and tend to be long as seen in Listing 4.9.

Chapter 5

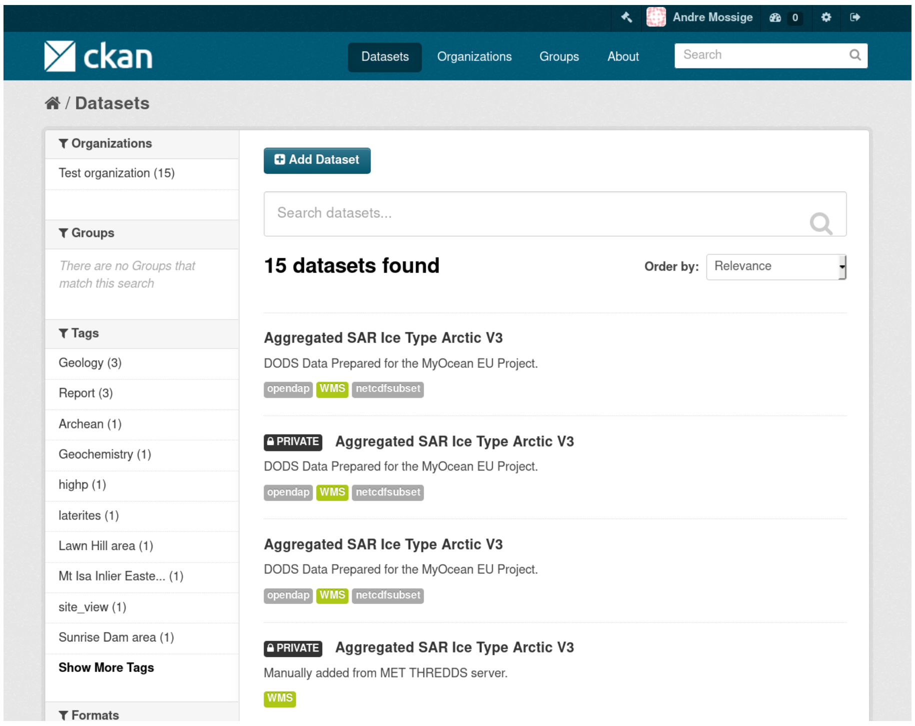
System overview and demonstration

In this chapter we will demonstrate the implemented GIS prototype. The use cases from Section 2.3.1 will be carried out and illustrated. We will examine the problems from Section 2.3.2, and evaluate how or if they are improved in Chapter 6. Some example datasets have been harvested from various external locations for the prototype to contain some content.

5.1 Finding data

The first use case is to find data. On the home page of the prototype (Figure 4.1), the user is immediately presented with an easy accessible search bar. If new datasets have been published recently, these are listed on the front page for easy navigation. We can enter the separate dataset page by clicking on "Datasets" at the top of the screen.

The dataset page is demonstrated in Figure 5.1. On this page all datasets available to a specific user is listed. The search bar lets us search the database using any words contained in the dataset fields. For example if we search for "sea ice", we will return datasets consisting of sea ice data. Figure 5.2 shows the search results. On the left we can further refine our search by selecting one or several filters. We can exclude datasets based on filters like owner organisations, groups, tags, formats and licenses.


Andre Mossige 0

ckan

Datasets Organizations Groups About Search

/ Datasets

Organizations

Test organization (15)

Groups

There are no Groups that match this search

Tags

Geology (3)

Report (3)

Archean (1)

Geochemistry (1)

highp (1)

laterites (1)

Lawn Hill area (1)

Mt Isa Inlier Easte... (1)

site_view (1)

Sunrise Dam area (1)

Show More Tags

Formats

Add Dataset

Search datasets...

15 datasets found Order by: Relevance

Aggregated SAR Ice Type Arctic V3

DODS Data Prepared for the MyOcean EU Project.

opendap WMS netcdfsubset

PRIVATE **Aggregated SAR Ice Type Arctic V3**

DODS Data Prepared for the MyOcean EU Project.

opendap WMS netcdfsubset

Aggregated SAR Ice Type Arctic V3

DODS Data Prepared for the MyOcean EU Project.

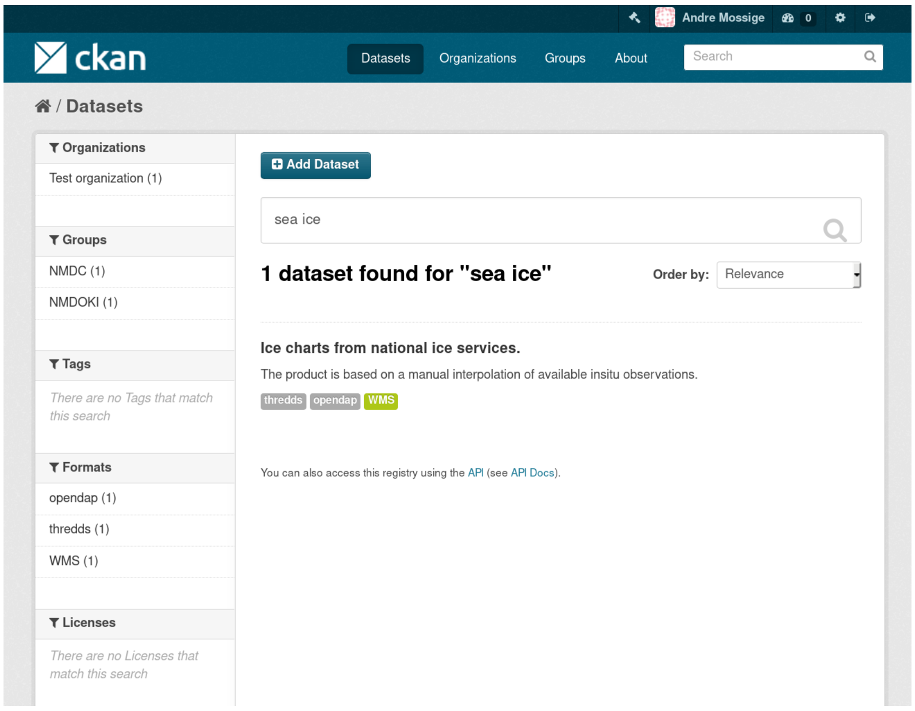
opendap WMS netcdfsubset

PRIVATE **Aggregated SAR Ice Type Arctic V3**

Manually added from MET THREDDS server.

WMS

Figure 5.1: Prototype dataset page


Andre Mossige 0

ckan

Datasets Organizations Groups About Search

/ Datasets

Organizations

Test organization (1)

Groups

NMDC (1)

NMDOKI (1)

Tags

There are no Tags that match this search

Formats

opendap (1)

thredds (1)

WMS (1)

Licenses

There are no Licenses that match this search

Add Dataset

sea ice

1 dataset found for "sea ice" Order by: Relevance

Ice charts from national ice services.

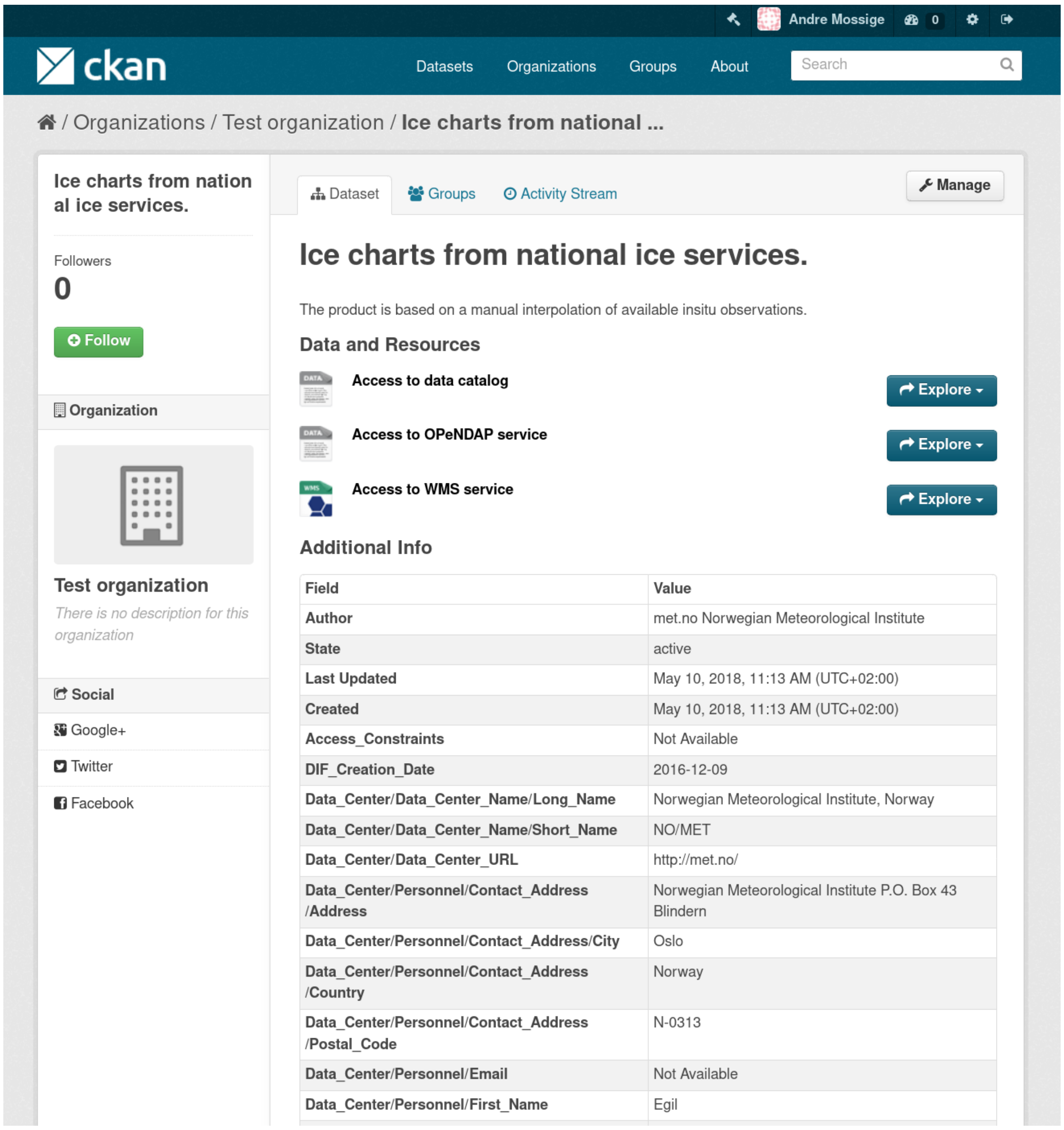
The product is based on a manual interpolation of available insitu observations.

thredds opendap WMS

You can also access this registry using the API (see API Docs).

Figure 5.2: Searching for a dataset containing sea ice data

Clicking on the dataset title leads us to the dataset details page, as shown in Figure 5.3. Here we can examine the resources and the metadata of the dataset. Some of the displayed metadata fields are redundant, and some can be joined together. This is an area that can be improved in future versions of the prototype. Clicking on "Access to WMS service" brings us to the page shown in Figure 5.4.



Ice charts from national ice services.

The product is based on a manual interpolation of available insitu observations.

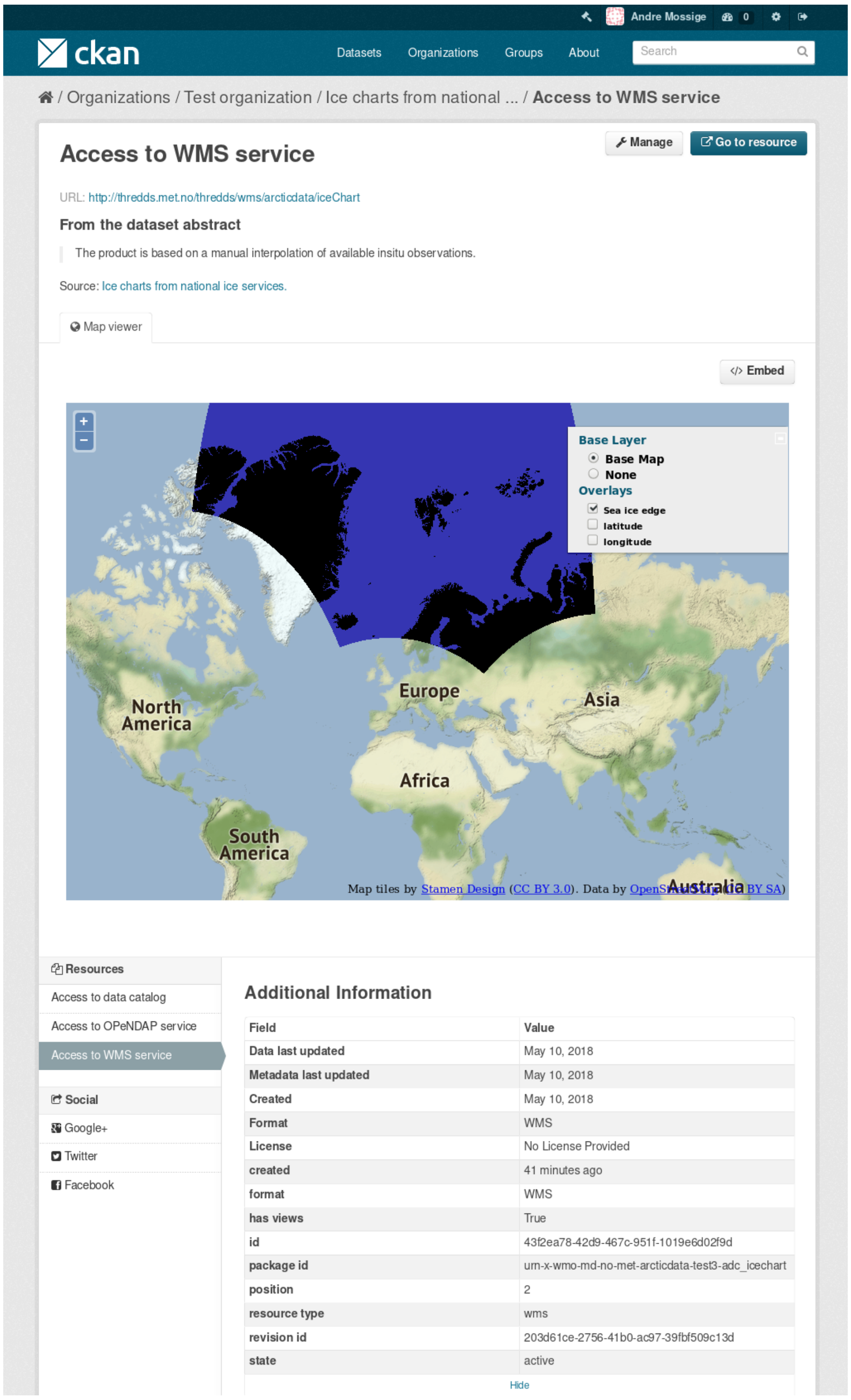
Data and Resources

- Access to data catalog — Explore
- Access to OPeNDAP service — Explore
- Access to WMS service — Explore

Additional Info

Field	Value
Author	met.no Norwegian Meteorological Institute
State	active
Last Updated	May 10, 2018, 11:13 AM (UTC+02:00)
Created	May 10, 2018, 11:13 AM (UTC+02:00)
Access_Constraints	Not Available
DIF_Creation_Date	2016-12-09
Data_Center/Data_Center_Name/Long_Name	Norwegian Meteorological Institute, Norway
Data_Center/Data_Center_Name/Short_Name	NO/MET
Data_Center/Data_Center_URL	http://met.no/
Data_Center/Personnel/Contact_Address/Address	Norwegian Meteorological Institute P.O. Box 43 Blindern
Data_Center/Personnel/Contact_Address/City	Oslo
Data_Center/Personnel/Contact_Address/Country	Norway
Data_Center/Personnel/Contact_Address/Postal_Code	N-0313
Data_Center/Personnel/Email	Not Available
Data_Center/Personnel/First_Name	Egil

Figure 5.3: *Ice charts from national ice services* dataset page



/ Organizations / Test organization / Ice charts from national ... / **Access to WMS service**

Access to WMS service

URL: http://thredds.met.no/thredds/wms/arcticdata/iceChart

From the dataset abstract

> The product is based on a manual interpolation of available insitu observations.

Source: Ice charts from national ice services.

Base Layer: [✓ Base Map] None

Overlays: [✓ Sea ice edge] latitude longitude

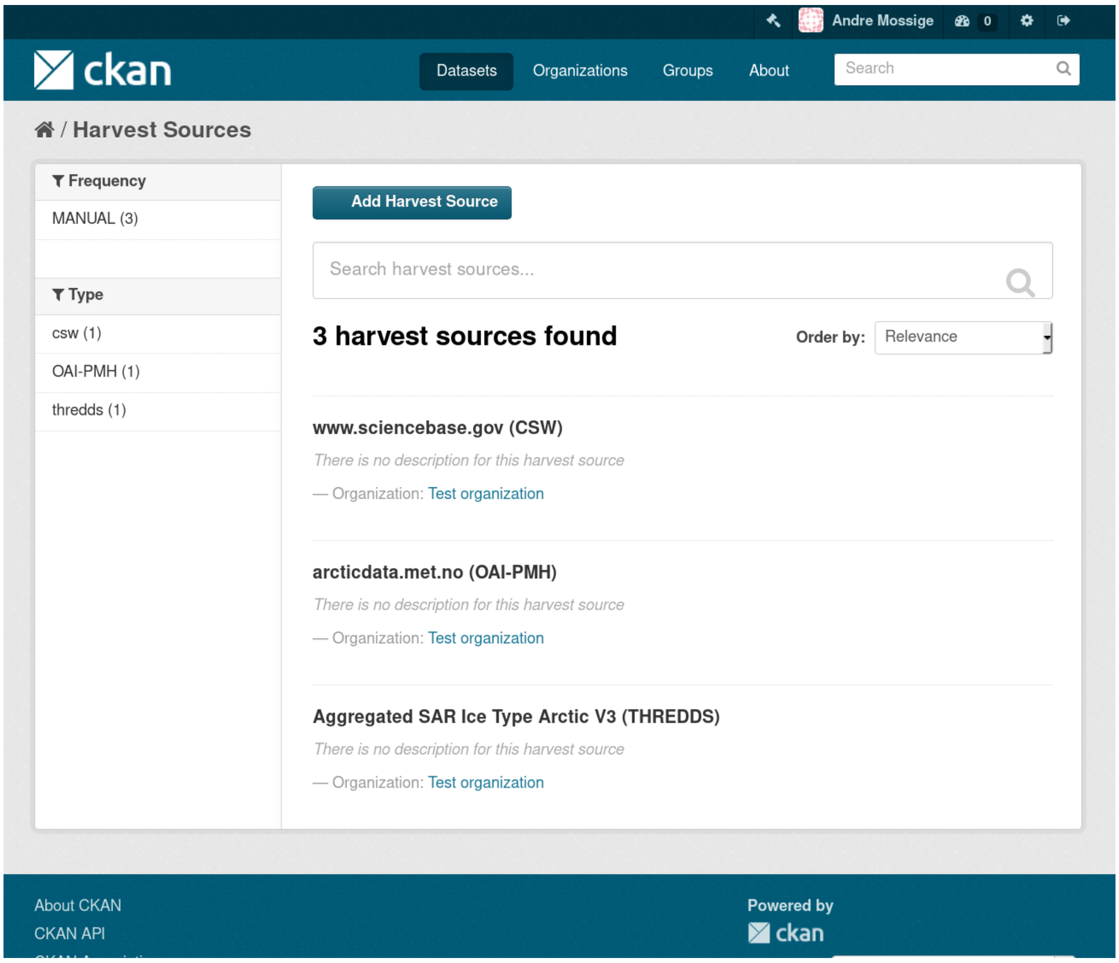
Additional Information

Field	Value
Data last updated	May 10, 2018
Metadata last updated	May 10, 2018
Created	May 10, 2018
Format	WMS
License	No License Provided
created	41 minutes ago
format	WMS
has views	True
id	43f2ea78-42d9-467c-951f-1019e6d02f9d
package id	urn-x-wmo-md-no-met-arcticdata-test3-adc_icechart
position	2
resource type	wms
revision id	203d61ce-2756-41b0-ac97-39fbf509c13d
state	active

Figure 5.4: Preview of WMS service

5.2 Adding a new harvesting source

We have not implemented a separate button for entering the page that lets us view and add new harvest sources. This should ideally be included in a future version. By navigating to the URL `http://<host-domain>/harvest`, we get an overview of the added harvest sources. The page is shown Figure 5.5.



3 harvest sources found

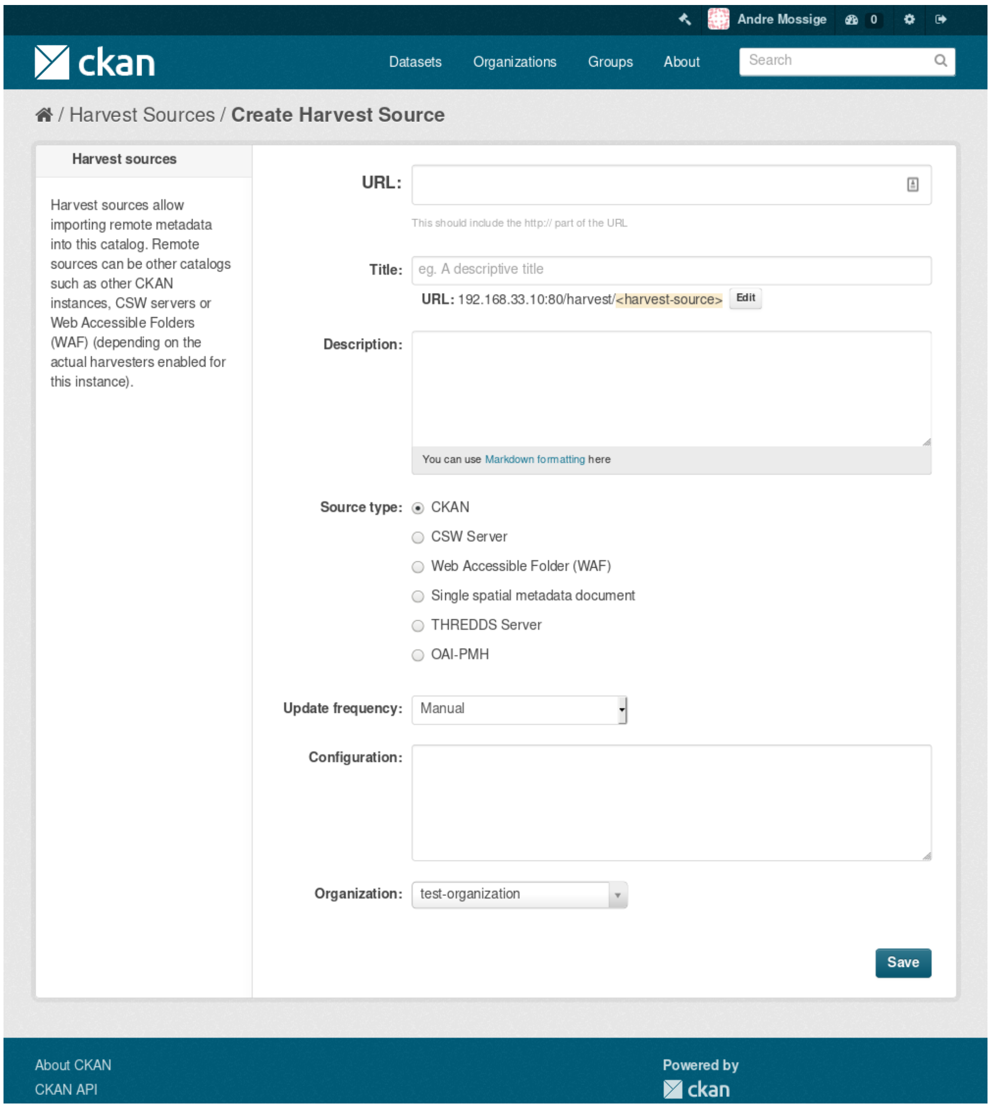
www.sciencebase.gov (CSW)
There is no description for this harvest source
— Organization: Test organization

arcticdata.met.no (OAI-PMH)
There is no description for this harvest source
— Organization: Test organization

Aggregated SAR Ice Type Arctic V3 (THREDDS)
There is no description for this harvest source
— Organization: Test organization

Figure 5.5: List of harvest sources

Clicking on the "Add Harvest Source" button brings us to the form to create a new harvest source. This is demonstrated in Figure 5.6. On this page the user have to fill in the necessary information. The most important information is the source URL and type. The list of supported harvest types are shown based on the harvesters that are enabled in the prototype. Our implemented harvesters are listed at the bottom: *THREDDS Server* and *OAI-PMH*. The *configuration* field lets the user pass settings to the different harvesters, to customise its operation and functionality. We have not implemented support for any customisations in our harvesters. This could be done in future versions. For example a field to manually choose the metadata format the OAI-PMH harvester should use could be useful. The user can also decide how often the source should be harvested with the *Update frequency* field. For example it can be harvested every day, every month or by manually clicking the "Reharvest button". When "Save" is clicked, the harvest source is saved to the database, ready to be harvested. When datasets are successfully imported, they will be displayed in the dataset list described in Section 5.1.



/ Harvest Sources / **Create Harvest Source**

Harvest sources

Harvest sources allow importing remote metadata into this catalog. Remote sources can be other catalogs such as other CKAN instances, CSW servers or Web Accessible Folders (WAF) (depending on the actual harvesters enabled for this instance).

URL:

This should include the http:// part of the URL

Title: eg. A descriptive title

URL: 192.168.33.10:80/harvest/<harvest-source> Edit

Description:

You can use Markdown formatting here

Source type: [✓ CKAN] CSW Server / Web Accessible Folder (WAF) / Single spatial metadata document / THREDDS Server / OAI-PMH

Update frequency: Manual

Configuration:

Organization: test-organization

Save

Figure 5.6: Page to add a new harvest source

5.3 Filter data before retrieval

Filtering and restricting a dataset before retrieval is enabled by OPeNDAP. We navigate to the dataset we are interested in, and if the dataset is accessible by OPeNDAP, we click "Access to OPeNDAP service". We are then redirected to the OPeNDAP access form (described in Section 3.3.2) where we can modify the access query.

This operation has room for improvement. Just like in some other GIS, the user is redirected back and forth between considerably different looking pages before they get access to the data. An improvement could be to implement the OPeNDAP access form in the prototype user interface itself. Instead of being redirected to a separate page, the user could modify the query in the prototype, and would perform the query and get a response. This is possible since the OPeNDAP access form simply updates an URL based on the entered data that the user can use. The same could be achieved within the prototype. We will come back to this improvement in Chapter 6 and 7.

5.4 Download data

Downloading data can be performed several ways. For example resources harvested from TDS often provide URLs pointing to FTP servers. This link can be clicked on directly from the prototype user interface, and a download will start. If the size of the resource data is several gigabytes or terabytes, users often want to use OPeNDAP to download parts of a dataset, as explained in the previous section. The URL pointing to the OPeNDAP service could also be copied by the user and pasted in their preferred desktop application, if it supports it.

5.5 Summary

All use cases given in Section 2.3.1 are implemented. The user can search for data, filter it with OPeNDAP, download it, and visualise it with WMS.

Chapter 6

Evaluation

One of the last steps of the research methodology of Peffers et al. (2007) [92] is to evaluate an artifact: our prototype. The evaluation will form a basis for future work. We will examine if our prototype solves the problems identified in Section 2.3.2, and compare the observed results from Chapter 5 with the objectives of the thesis given in Section 1.3.1. The research questions from Section 1.4 are then answered. We conclude the chapter by reviewing the overall goal of the thesis.

6.1 Sub-goals

We begin by evaluating the achievement of the sub-goals.

6.1.1 Overview of standards to build prototype

We performed an interview with two researchers to determine the data sources they use. We then researched what standards and protocols the data sources used to allow harvesting of them. We found that DIF and Dublin Core were appropriate for metadata. We used CSW, OAI-PMH and TDS for harvesting the metadata records. WMS and OPeNDAP was used for access to the geospatial datasets.

Steiniger and Hunter (2011) [94] describes several additional technical standards and protocols that are important for distribution of spatial data. For example Web Feature Service (WFS), WCS, ISO 19115 and ISO 19119 are mentioned. Due to time constraints, it was unfeasible to implement support for these additional standards. Because of CK-ANs flexible architecture, it would be possible to develop extensions that would allow

CKAN to support these standards. Such task is appropriate for future work.

6.1.2 Types of data supported in prototype

By interviewing the two researchers from NERSC we found that harvesting metadata from OAI-PMH repositories in Dublin Core and DIF, and harvesting TDS was sufficient for the thesis. The researchers used other data from other repositories as well. These repositories did not support access via OAI-PMH or TDS. Due to time constraints, we focused on implementing harvesting with the mentioned standards only. In retrospect, we could have interviewed more researchers to get a more realistic view of the types of data that are used the most.

6.1.3 GIS alternatives as prototype foundation

Amorim et al. (2017) [73] examined software for use as a research content management system. A number of GIS were discussed. The three most popular alternatives from the paper were considered. Amorim et al. recommended CKAN for many reasons, as outlined below.

Because CKAN is open-source and has an active developer community, it is easy to get help to solve problems. CKAN is used by many government institutions, and therefore gives a good indication that it is well suited for our prototype. The architecture of CKAN is flexible with a solid extension mechanism and open API. Hence, we can easily add extra functionality that is needed. Last, CKAN is written in Python, a high-level language that is relatively easy to use. For these reasons, CKAN was chosen to be used as the foundation for our prototype.

6.1.4 Set up a catalogue service

The objective of this sub-goal was to install and configure a metadata catalogue service using CKAN as a foundation. Since CKAN is not directly usable for storing geospatial metadata by default, extra features have to be supported by extensions. We installed three extensions: `ckanext-spatial`, `ckanext-harvest` and `ckanext-geoview`.

After installing these extensions, some of the prototype requirements were already fulfilled. For example users were able to search for datasets with the built in search functionality. The functionality were enabled by Solr. The search lets the user search for words appearing in the metadata of all datasets.

We mentioned in Section 4.3.1 that the spatial extension would allow users to search for datasets based on their geographic location. However, after further inspection, we did not finish all steps needed to enable this type of search operation. A special spatial field defining the bounding box of some dataset had to be manually added to the dataset metadata for this to work. We did not realise this until the end of the thesis work. Therefore this functionality is not finished. This should be addressed in a future version.

We experienced that some parts of the CKAN documentation was confusing. For example several guides exist for installing CKAN, but it is not apparent at first which one to follow when we want to develop extensions. Since CKAN is a relatively complex system, the documentation is extensive. The documentation is scattered around, and is sometimes hard to keep track of where to look for information. However, after working with the documentation and becoming familiar with it, it is relatively easy to use.

Some support for harvesting metadata were provided by the harvesting extension. For example importing datasets using CSW was possible. However, harvesting from OAI-PMH and TDS had to be implemented in our own extensions.

6.1.5 Implement harvesting capabilities

Developing extensions for CKAN was a generally pleasant experience. The objective of this sub-goal was to implement capabilities to harvest external data sources. This was achieved by developing two extensions: one to harvest OAI-PMH repositories, and one to harvest records from TDS. The result is that researchers can potentially use our prototype to find needed data without having to search using many different GIS. This will improve the time needed for researchers to search for data.

The `IHarvester` interface we had to implement for custom harvesters was hard to understand at first. For example knowing the parameter types we had available in the methods and what fields they contained was badly documented. We therefore spent a considerable amount of time using Python to print to the command line both the type and value of the parameter fields to be able to understand and implement all functionality.

We had some problems with metadata records not always conforming to their metadata standard. For example sometimes mandatory fields were missing. We had a similar problem with data sources using different versions of metadata standards. In some versions some fields are mandatory, and others are not.

The TDS crawler was not as robust as we would hope. For example sometimes the OPeNDAP URL for a resource did not work. When we tried to access them via the prototype user interface we are presented with a *"404 not found"* error for some resources. This error might be related to the problem of resources not providing us with the expected fields. This should be corrected in future versions.

Of the data sources mentioned by researchers, only CSW, TDS and OAI-PMH can be harvested. We have not implemented harvesting from NIRD. Harvesting from this repository should be implemented in future work.

6.1.6 Implement simple geospatial visualisation

Simple visualisation of for example WMS were supported out of the box with `ckanext-geoview`. The visualisation is very simple, and can be improved. For example there is no legend that describe the various layers of a dataset, except displaying the names of the layers. This could be improved in future work. Additionally, it is sometimes hard to distinguish between the different layers, as their colour and opacity are not adjustable.

6.1.7 Evaluation of prototype

The aim of this sub-goal was to evaluate the finished prototype. The evaluation is done in this chapter. We conclude that all requirements from Section 2.1.3 are implemented to some or full extent:

- **Support most widely used standards and protocols**

  The reader is referred to Section 6.1.1 for status.

- **Searching for datasets**

  Searching is provided by Solr. A user can search for words that appear in the metadata of datasets. We started developing search based on geographic area, which is not fully implemented at this stage.

- **Harvesting capabilities for the most used standards**

  We implemented support for harvesting OAI-PMH repositories in Dublin Core and DIF and harvesting TDS. Refer to Section 6.1.5 and Section 7.2.7 for details.

- **Simple spatial visualisation**

  The prototype can visualise resources using WMS. This requirement is fully implemented.

- **Downloading datasets**

  Downloading of datasets are provided via FTP and OPeNDAP. This is implemented per the requirement. Ideas for further improvements are given in Section 7.2.5.

6.2 Research questions

We try to answer the research questions defined in Chapter 1. The questions address research topics related to the overall and specific goals of the thesis. The questions guided the thesis work by keeping the goal of the thesis clear: simplifying the discovery of research data. The discussion below aims to answer the research questions, and gives links to relevant literature.

- **How can a GIS that combines different data and formats within the system, aid a researcher in doing research?**

  A GIS that harvests external data repositories help researchers in several ways. Only having one system to use simplifies the process of searching for data. The researchers do not have to manually search every possible source for data. Additionally, they do not have to spend time to get used to the different user interfaces that different data sources use. Researchers can then spend more time doing actual research, than to look for data to use in it.

  When researchers have to use several GIS to find their data, the various systems often provide exporting methods with distinct download formats. If only having a single GIS to use, that system could give options to convert a dataset to many formats before download [95]. This would give a more standardised experience to the user by giving the opportunity to export the data in the formats needed.

  A single GIS could in a similar fashion provide processing for datasets before exporting it. This could be helpful by for example cleaning up a dataset before download, removing unneeded parameters [97].

- **How should a systems software architecture be designed to allow extending its functionality?**

  Developing software that has a flexible architecture allowing for extending is a whole research area in itself [81, 76]. We will answer the research question considering how *CKANs* architecture allows for extensibility. The CKAN developers do not recommend extending CKAN by modifying its source code directly. Instead it lets code that is separated from the source code hook into most parts of CKAN. We can modify CKAN by accessing the CKAN business logic using its open API, as well as implementing provided interfaces and inheriting from existing classes. This way, functionality can be changed in a controlled way, compared to changing the source code directly. The modular architecture ensures low coupling and high cohesion, since extensions are kept separate from CKAN itself and only communicates with the CKAN source code when needed.

6.3 Overall goal

We have successfully developed a GIS prototype that harvests metadata from external resources. It supports harvesting data repositories with most commonly used standards. These include CSW, OAI-PMH and TDS. The harvested datasets are searchable, and can be visualised using WMS. Researchers at NERSC also use data from repositories not allowing harvesting using any of the implemented harvesting methods. For example NIRD can not be harvested in the prototype. For the prototype to be improved, harvesting from this repository should be developed.

We have mentioned the achievement of some of the use cases from Section 2.3.1 in the previous sections. Finding and searching for data, filtering data with OPeNDAP, and downloading it with OPeNDAP and FTP is supported in the prototype. We will discuss some of these use cases further in Section 7.2.

Some of the expressed problems with GIS given from users in Section 2.3.2 are improved with our prototype. Having to deal with many web sites to find needed data is partly solved. Instead of searching for data in both CMIP and NMDC, users can search within our prototype only. Users still have to use other web sites to search for data from NIRD. The problem of being redirected to other web sites for download and access of actual data is partly improved. Users are still being redirected to the OPeNDAP access form to download data. Download via FTP works without being redirected, since users click the FTP link in our prototype and a download is immediately started. Verifying correct dataset using visualisations before download is also partially improved. The prototype can visualise data with WMS, and display the different layers a dataset contains. However, the core of the problem is related to visualising a dataset *after* restricting it with OPeNDAP. This is not supported in the prototype. We will come back to the unsolved problems in Section 7.2

Chapter 7

Conclusion

This last chapter concludes our thesis. We give a summary of the current status of our GIS prototype and thesis work. We then list ideas for improvements that would have been implemented if we have had more time. These tasks should be considered for future work. Last, we give a final conclusion to the overall thesis, what we have achieved and what the author has learnt.

7.1 Summary of results

A GIS prototype that can harvest metadata from external data sources is developed. We began by selecting a number of standards and protocols commonly used in the geoscientific field. By reviewing their specifications, we learnt about how the different standards function. Different GIS frameworks were considered, and we determined CKAN to be the best foundation for our prototype. We built competence in CKAN, especially the extension mechanism. Next we set up and configured the default CKAN installation, and installed three extensions to enable geospatial capabilities.

We implemented two extensions, one that can harvest data from OAI-PMH repositories, and one to harvest TDS. The prototype supports searching for harvested data, simple visualisations of the data using WMS, and downloading the data with FTP or OPeNDAP.

In the thesis we have documented the process of developing the prototype. The finished prototype was demonstrated and evaluated. We evaluated the prototype by reviewing the sub-goals defined for the thesis. We also studied relevant literature and answered the research questions.

7.2 Future work

The GIS prototype can be improved. In this section we will explore some of the possible areas that could be considered for future work, and explain how we would have approached a solution. Accurately estimating the amount of work a programming task requires is complicated and challenging [85]. We will try to give a rough indication of the amount of work required to implement the following tasks. These estimations are based on our own experience with CKAN and its extension mechanism. A developer addressing these features would need similar experience with CKAN, relevant GIS standards and Python knowledge.

7.2.1 Perform user testing

We planned to perform user testing of the prototype. Due to time constraints we evaluated the prototype from our own perspective instead. Performing user testing on real users would provide us with real feedback and evaluation. The feedback would be more valuable than our own evaluation. This should be a priority for future work. Testing could include collecting metrics of common tasks. For example measuring the time it takes for a user to find a certain dataset. The metrics collected from the prototype should be compared with metrics from current GIS. This would give evidence on the performance of the prototype.

7.2.2 Clean up harvested metadata for display

In the current prototype we directly display metadata returned from harvested repositories. This means that for example metadata fields with identical values are displayed several times. For example in DIF the fields `Personnel/Role` sometimes appear several times with the same values. The redundant fields should be removed from the user interface. Other fields could also be combined for better visibility. For example the fields `first_name` and `last_name` do not have to be displayed separately. The list of metadata fields are sorted alphabetically, meaning that the first name and last name will not be displayed next to each other.

There appears to be several ways to approach the development of this feature. For example there is an interface `IResourceView` that could be implemented for all the different resources that we harvest. In the interface we would define how a specific resource would be displayed. Another approach would be to implement another interface `IDatasetForm`, that would allow extra metadata field to be displayed. The advantage to this interface is that it includes methods for validating the metadata fields before display. In general this feature should not take long to implement, maybe one or two days.

7.2.3 Improve display of TDS links harvested with OAI-PMH

Occasionally resources harvested from an OAI-PMH repository only provide access via a link to a TDS catalogue. This catalogue may contain further links to OPeNDAP, WMS or FTP. The consequence is that the user have to navigate between different web pages before they can access the data.

This could be fixed in an implementation similar as in the previous problem. For example a special resource view for TDS links could be created with the `IResourceView` interface. Logic for fetching the access links from the TDS data catalogue would have to be developed. The amount of work required for this task should take longer than the previous one. Some additional logic is required to harvest the TDS catalogue before display. Some challenging corner cases might come up while implementing the feature. For example what to do if the TDS link points to several datasets, and not only one.

7.2.4 Improve logic for harvesting periodic measurements

TDS catalogues are often used for storing periodic measurements. For example a single folder in a TDS catalogue could contain several hundred resources. Harvesting all of these resources for display in the prototype might not be necessary. A better solution might be to only harvest the single aggregated resource with all the periodic measurements combined.

This task could be completed in a few days time. The `thredds_crawler` contains logic for choosing what resources should be harvested based on a regular expression. This expression could be used to ignore resources that do not contain the word "Aggregated". This word is used for the aggregated dataset that TDS creates when automatically combining periodic measurements. A boolean option in the prototype user interface to create new harvest sources could be added to let the user choose if all the separate periodic resources should be harvested individually or not.

7.2.5 Display OPeNDAP access form directly in prototype

If users want to access a dataset using OPeNDAP they have to navigate to the OPeNDAP access form using the link displayed in the prototype. Researchers expressed that having to navigate between many web pages were annoying. We could interview researchers and ask if this is a feature that they would want. If so, a separate resource view for OPeNDAP access forms could be implemented with the `IResourceView` interface. The result would be that users do not have to navigate to the separate OPeNDAP web page first. The access form would be displayed directly in our prototype instead. Implementing this view would require more work than previous tasks. We estimate it would take around a couple of weeks to finalise an OPeNDAP access form.

7.2.6 Improve detection of resource formats

We mentioned in Chapter 4 that we used a simplistic approach when detecting types of resources harvested from OAI-PMH repositories. Currently we check the resource URLs to determine its type. For example if the URL contains “wms” then it is a WMS service. This could be improved to be more robust. The amount of work required for this task should be relatively little. We estimate it would take a couple of days using the following solution.

For example in DIF, additional information is provided that could be used to determine type. The fields `Type`, `SubType` and `Description` could be used. For example the `Type` field can contain “OPENDAP DATA (DODS)”, giving a better indication that the resource is to an OPeNDAP access form. This is more robust as the resource URL can now change without our format detection breaking.

7.2.7 Implement harvesting from more data sources

We have only implemented harvesting from TDS and OAI-PMH. Support for a number of other data sources should be developed for the GIS to be usable. For example in the interview with researchers from NERSC, NIRD was a data source that was also being used daily.

Research should be done beforehand to check if it is possible to harvest these data sources. For example if they provide any open APIs, or if they use any open-source standards allowing harvesting from them. If so, a new harvesting extension could be developed. If not, it might not be possible to harvest from these sources. The next step could then be to contact the maintainers or developers of the data sources and start a dialog about opening up access to them.

Considerably amount of work would be required to implement harvesters for new types of data sources. Amount of work would differ for various types of data sources, but at least a few weeks per data source.

7.2.8 General improvements to harvesters

Last, we briefly mention some general improvements that should be done to the harvesters we implemented. For the prototype to be usable, the harvester processes should be managed by a process manager like *Supervisor*. Right now they are started manually from the command line. By using Supervisor, the jobs would be running constantly and harvest data sources.

Better error handling should also be implemented, so that the code is easier to debug when errors occur. Simple error handling, or no error handling at all is currently implemented. The code should be refactored, moving related logic into own methods and classes to make it more readable. Additionally, more tests should be written to verify the correctness of the code.

7.3 Conclusion

We have created a functional GIS prototype that can harvest metadata records from various external data servers. The prototype helps researchers by simplifying the process of finding research data. The prototype is still a work in progress. Support for harvesting additional data servers and other implementation improvements are necessary.

When developing a GIS that harvests metadata, we assume that repositories always provide a way for us to access their data. This is a radical assumption, and is not always true. Once there is a data source that do not provide means of harvesting from it, researchers potentially have to use several GIS to find their data anyway. This is a problem of standardisation: it is challenging, or impossible, to get everyone to agree on using the same standards. This is because different requirements exist for different problems, resulting in a need for different standards. A solution to this problem is outside the scope of the thesis.

The research and development process has been a learning process for the author. GIS software has been a completely new field, and learning how the relevant technical standards and protocols work have been a challenge. The specifications for these are very technical, and are hard to read. Building knowledge in an existing code base written in a new programming language and reading documentation to solve problems have been engaging. During the implementation process the author has become more knowledgeable in Python.

Appendix A

Code listings

A.1 IHarvester interface

The `IHarvester` interface describes methods that CKAN extensions must implement to perform harvesting operations. The details of `IHarvester` can found at the GitHub page of `ckanext-harvest` [16].

```
from ckan.plugins.interfaces import Interface

class IHarvester(Interface):
    '''
    Common harvesting interface
    '''

    def info(self):
        '''
        :returns: A dictionary with the harvester descriptors
        '''

    def validate_config(self, config):
        '''
        :param harvest_object_id: Config string coming from the form
        :returns: A string with the validated configuration options
        '''

    def get_original_url(self, harvest_object_id):
        '''
        [optional]
        :param harvest_object_id: HarvestObject id
        :returns: A string with the URL to the original document
        '''

    def gather_stage(self, harvest_job):
        '''
        :param harvest_job: HarvestJob object
        :returns: A list of HarvestObject ids
        '''

    def fetch_stage(self, harvest_object):
        '''
        :param harvest_object: HarvestObject object
        :returns: True if successful, 'unchanged' if nothing to import ↩
            after
                  all, False if not successful
        '''

    def import_stage(self, harvest_object):
        '''
        :param harvest_object: HarvestObject object
        :returns: True if the action was done, "unchanged" if the object ↩
            didn't
                  need harvesting after all or False if there were errors↩
                  .
        '''
```

Listing A.1: IHarvester interface

A.2 CKAN configuration file

`production.ini` contains global configuration options for CKAN. For example it specifies the locations of the PostgreSQL database and Solr. All configuration options available is found in the CKAN documentation [10].

```
#
# CKAN - Pylons configuration
#
# These are some of the configuration options available for your CKAN
# instance. Check the documentation in 'doc/configuration.rst' or at the
# following URL for a description of what they do and the full list of
# available options:
#
# http://docs.ckan.org/en/latest/maintaining/configuration.html
#
# The %(here)s variable will be replaced with the parent directory of ↩
    this file
#

[DEFAULT]

```

```
# WARNING: *THIS SETTING MUST BE SET TO FALSE ON A PRODUCTION ENVIRONMENT*
debug = false

[server:main]
use = egg:Paste#http
host = 0.0.0.0
port = 5000

[app:main]
use = egg:ckan
full_stack = true
cache_dir = /tmp/%(ckan.site_id)s/
beaker.session.key = ckan

# This is the secret token that the beaker library uses to hash the cookie sent
# to the client. `paster make-config` generates a unique value for this each
# time it generates a config file.
beaker.session.secret = gf58Qr7VeoJ7W100ZU+/sPejh

# `paster make-config` generates a unique value for this each time it generates
# a config file.
app_instance_uuid = dc90e53b-c055-417f-8d6c-1439d00a3f91

# repoze.who config
who.config_file = %(here)s/who.ini
who.log_level = warning
who.log_file = %(cache_dir)s/who_log.ini
# Session timeout (user logged out after period of inactivity, in seconds).
# Inactive by default, so the session doesn't expire.
# who.timeout = 86400

## Database Settings
sqlalchemy.url = postgresql://ckan_default:pass@localhost/ckan_default

#ckan.datastore.write_url = postgresql://ckan_default:pass@localhost/datastore_default
#ckan.datastore.read_url = postgresql://datastore_default:pass@localhost/datastore_default

# PostgreSQL' full-text search parameters
ckan.datastore.default_fts_lang = english
ckan.datastore.default_fts_index_method = gist

## Site Settings

ckan.site_url = http://192.168.33.10:80
#ckan.use_pylons_response_cleanup_middleware = true

```

```
## Authorization Settings

ckan.auth.anon_create_dataset = false
ckan.auth.create_unowned_dataset = false
ckan.auth.create_dataset_if_not_in_organization = false
ckan.auth.user_create_groups = false
ckan.auth.user_create_organizations = false
ckan.auth.user_delete_groups = true
ckan.auth.user_delete_organizations = true
ckan.auth.create_user_via_api = false
ckan.auth.create_user_via_web = true
ckan.auth.roles_that_cascade_to_sub_groups = admin


## Search Settings

ckan.site_id = default
solr_url = http://127.0.0.1:8983/solr


## Redis Settings

# URL to your Redis instance, including the database to be used.
ckan.redis.url = redis://localhost:6379/0
# Backend for ckanext-harvest
ckan.harvest.mq.type = redis


## CORS Settings

# If cors.origin_allow_all is true, all origins are allowed.
# If false, the cors.origin_whitelist is used.
# ckan.cors.origin_allow_all = true
# cors.origin_whitelist is a space separated list of allowed domains.
# ckan.cors.origin_whitelist = http://example1.com http://example2.com


## Plugins Settings

# Note: Add ``datastore`` to enable the CKAN DataStore
#       Add ``datapusher`` to enable DataPusher
#               Add ``resource_proxy`` to enable resorce proxying and get↩
    around the
#               same origin policy
ckan.plugins = stats text_view image_view recline_view resource_proxy ↩
    geo_view spatial_metadata spatial_query harvest ckan_harvester ↩
    csw_harvester doc_harvester waf_harvester thredds_harvester ↩
    oaipmh_harvester

# Define which views should be created by default
# (plugins must be loaded in ckan.plugins)
ckan.views.default_views = image_view text_view recline_view geo_view

```

```
# Customize which text formats the text_view plugin will show
#ckan.preview.json_formats = json
#ckan.preview.xml_formats = xml rdf rdf+xml owl+xml atom rss
#ckan.preview.text_formats = text plain text/plain

# Customize which image formats the image_view plugin will show
#ckan.preview.image_formats = png jpeg jpg gif

## Front-End Settings
ckan.site_title = CKAN
ckan.site_logo = /base/images/ckan-logo.png
ckan.site_description =
ckan.favicon = /base/images/ckan.ico
ckan.gravatar_default = identicon
ckan.preview.direct = png jpg gif
ckan.preview.loadable = html htm rdf+xml owl+xml xml n3 n-triples turtle ↩
    plain atom csv tsv rss txt json
ckan.display_timezone = server

# package_hide_extras = for_search_index_only
#package_edit_return_url = http://another.frontend/dataset/<NAME>
#package_new_return_url = http://another.frontend/dataset/<NAME>
#ckan.recaptcha.version = 1
#ckan.recaptcha.publickey =
#ckan.recaptcha.privatekey =
#licenses_group_url = http://licenses.opendefinition.org/licenses/groups/↩
    ckan.json
# ckan.template_footer_end =


## Internationalisation Settings
ckan.locale_default = en
ckan.locale_order = en pt_BR ja it cs_CZ ca es fr el sv sr sr@latin no sk↩
     fi ru de pl nl bg ko_KR hu sa sl lv
ckan.locales_offered =
ckan.locales_filtered_out = en_GB

## Feeds Settings

ckan.feeds.authority_name =
ckan.feeds.date =
ckan.feeds.author_name =
ckan.feeds.author_link =

## Storage Settings

#ckan.storage_path = /var/lib/ckan
#ckan.max_resource_size = 10
#ckan.max_image_size = 2

## Datapusher settings

# Make sure you have set up the DataStore
```

```

#ckan.datapusher.formats = csv xls xlsx tsv application/csv application/vnd.ms-excel application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
#ckan.datapusher.url = http://127.0.0.1:8800/
#ckan.datapusher.assume_task_stale_after = 3600

# Resource Proxy settings
# Preview size limit, default: 1MB
#ckan.resource_proxy.max_file_size = 1048576
# Size of chunks to read/write.
#ckan.resource_proxy.chunk_size = 4096

## Activity Streams Settings

#ckan.activity_streams_enabled = true
#ckan.activity_list_limit = 31
#ckan.activity_streams_email_notifications = true
#ckan.email_notifications_since = 2 days
ckan.hide_activity_from_users = %(ckan.site_id)s


## Email settings

#email_to = errors@example.com
#error_email_from = ckan-errors@example.com
#smtp.server = localhost
#smtp.starttls = False
#smtp.user = username@example.com
#smtp.password = your_password
#smtp.mail_from =


## Logging configuration
[loggers]
keys = root, ckan, ckanext

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = WARNING
handlers = console

[logger_ckan]
level = INFO
handlers = console
qualname = ckan
propagate = 0

```

```
[logger_ckanext]
level = DEBUG
handlers = console
qualname = ckanext
propagate = 0

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic

[formatter_generic]
format = %(asctime)s %(levelname)-5.5s [%(name)s] %(message)s
```

Listing A.2: Global CKAN configuration file

A.3 ThreddsHarvester implementation

`ThreddsHarvester` contains the core logic of the TDS harvester we developed.

```python
from ckan.logic import get_action
from ckan.model import Session
from ckanext.harvest.model import HarvestObject, HarvestObjectExtra
from ckanext.harvest.harvesters import HarvesterBase
from thredds_crawler.crawl import Crawl
from hashlib import sha1
from datetime import datetime
import json
import uuid
import traceback


class ThreddsHarvester(HarvesterBase):

    def info(self):
        return {
            'name': 'thredds',
            'title': 'THREDDS Server',
            'description': 'A harvester that can read data and services '
            'from a THREDDS server.'
        }

    def gather_stage(self, harvest_job):
        print('THREDDS harvester gather_stage')
        c = Crawl(
                harvest_job.source.url,
                after = datetime.today())

        if not c.datasets:
            self._save_gather_error('No datasets found for %s' % ↩
                harvest_job.source.url)
            return []

        print('THREDDS crawler found %d datasets' % len(c.datasets))
        for dataset in c.datasets:
            ids = []
            # Generate GUID based on dataset id
            guid = sha1(dataset.id.encode()).hexdigest()
            package_dict = self._create_package_dict(dataset)
            content = json.dumps(package_dict)
            obj = HarvestObject(
                    guid=guid,
                    job=harvest_job,
                    content=content,
                    extras=[HarvestObjectExtra(key='status', value='new')↩
                        ])
            obj.save()
            ids.append(obj.id)

        return ids

    def fetch_stage(self, harvest_object):
        # Fetching already done in gather_stage.
        # thredds_crawler does not fetch IDs and content in different ↩
            stages.
        return True

    def import_stage(self, harvest_object):
        print('THREDDS harvester import_stage')
        try:
            package_dict = json.loads(harvest_object.content)
            self._update_current_object(harvest_object)
            self._set_owner_org(harvest_object, package_dict)

            # Save reference to the package of the object
            harvest_object.package_id = package_dict['id']
            harvest_object.add()

            Session.execute('SET CONSTRAINTS ↩
                harvest_object_package_id_fkey DEFERRED')
            Session.flush()

            self._create_or_update_package(
                package_dict,
                harvest_object
            )
            Session.commit()
            return True
        except:
            self._save_object_error(
                'Exception in import stage',
                harvest_object
            )
            traceback.print_exc()
            return False

    def _set_owner_org(self, harvest_object, package_dict):
        context = {
            'user': self._get_user_name(),
            'ignore_auth': True,
        }
        source_dataset = get_action('package_show')(
            context,
            {'id': harvest_object.source.id}
        )
        owner_org = source_dataset.get('owner_org')
        if owner_org:
            package_dict['owner_org'] = owner_org


    def _update_current_object(self, harvest_object):
        '''
        Get the last harvested object for this source and flag it
        as not current anymore.
        Flag new harvest object as current.
        '''
        previous_object = Session.query(HarvestObject) \
            .filter(HarvestObject.guid == harvest_object.guid) \
            .filter(HarvestObject.current is True) \
            .first()

        if previous_object:
            previous_object.current = False
            previous_object.add()

        # Flag this object as current
        harvest_object.current = True
        harvest_object.add()

    def _create_package_dict(self, dataset):
        rights = str(dataset.metadata.xpath("//*[name()='documentation'][↩
            @type='rights']/text()")[0])
        summary = str(dataset.metadata.xpath("//*[name()='documentation↩
            '][@type='summary']/text()")[0])
        author = str(dataset.metadata.xpath("//*[name()='publisher']/*[↩
            name()='name']/text()")[0])
        author_email = str(dataset.metadata.xpath("//*[name()='publisher↩
            ']/*[name()='contact']/@email")[0])

        package_dict = {}
        # We need to explicitly provide a package ID
        package_dict['id'] = str(uuid.uuid4())
        package_dict['name'] = package_dict['id']
        package_dict['title'] = dataset.name
        package_dict['notes'] = summary
        package_dict['license_id'] = rights
        package_dict['author'] = author
        package_dict['author_email'] = author_email

        self._add_resources(dataset, package_dict)
        self._add_extra_fields(dataset, package_dict)
        return package_dict

    def _add_resources(self, dataset, package_dict):
        formats = []
        package_dict['resources'] = []
        for service in dataset.services:
            resource = {
                'name': service['name'],
                'resource_type': service['service'].lower(),
                'format': service['service'].lower(),
                'url': service['url']
            }
            package_dict['resources'].append(resource)
            formats.append(service['service'].lower())
        package_dict['formats'] = formats

    def _add_extra_fields(self, dataset, package_dict):
        # Fields not in CKAN schema
        service_name = str(dataset.metadata.xpath("//*[name()='↩
            serviceName']/text()"))
        author_url = str(dataset.metadata.xpath("//*[name()='publisher↩
            ']/*[name()='contact']/@url")[0])
        data_format = str(dataset.metadata.xpath("//*[name()='dataFormat↩
            ']/text()")[0])

        extras = []
        extras.append(('Data format', data_format))
        extras.append(('Author URL', author_url))
        extras.append(('Service name', service_name))
        package_dict['extras'] = extras
```
Listing A.3: Implementation of ThreddsHarvester

Bibliography

[1] https://geo-ide.noaa.gov/wiki/index.php?title=Web_Accessible_Folder. Accessed: 29.05.18.

[2] About ckan. https://ckan.org/about/. Accessed: 12.04.18.

[3] Action api reference. http://docs.ckan.org/en/latest/api/index.html#api-reference. Accessed: 19.04.18.

[4] Apache hadoop homepage. https://hadoop.apache.org/. Accessed: 30.05.18.

[5] Ckan - exception handling. http://docs.ckan.org/en/ckan-2.7.3/extensions/tutorial.html#exception-handling. Accessed: 30.04.18.

[6] Ckan - the plugins toolkit. http://docs.ckan.org/en/ckan-2.7.3/extensions/tutorial.html#the-plugins-toolkit. Accessed: 30.04.18.

[7] Ckan code architecture. http://docs.ckan.org/en/latest/contributing/architecture.html. Accessed: 24.04.18.

[8] ckan-dev – ckan development discussions. https://lists.okfn.org/mailman/listinfo/ckan-dev. Accessed: 12.04.18.

[9] Ckan documentation - install and configure solr. http://docs.ckan.org/en/latest/maintaining/installing/install-from-package.html#install-and-configure-solr. Accessed: 29.05.18.

[10] Ckan documentation: Configuration options. http://docs.ckan.org/en/latest/maintaining/configuration.html. Accessed: 24.05.18.

[11] Ckan, download and install. https://ckan.org/download-and-install/. Accessed: 17.04.18.

[12] Ckan extensions. http://extensions.ckan.org/. Accessed: 12.04.18.

[13] Ckan github. https://github.com/ckan/ckan. Accessed: 09.05.18.

[14] Ckan github issues page. https://github.com/ckan/ckan/issues. Accessed: 12.04.18.

[15] ckanext-geoview - geospatial viewer for ckan resources. https://github.com/ckan/ckanext-geoview. Accessed: 12.04.18.

[16] ckanext-harvest - remote harvesting extension. https://github.com/ckan/ckanext-harvest. Accessed: 23.04.18.

[17] ckanext-spatial - geo related plugins for ckan. http://docs.ckan.org/projects/ckanext-spatial/en/latest/. Accessed: 18.04.18.

[18] Content management system definition. http://www.businessdictionary.com/definition/content-management-system-CMS.html. Accessed: 09.05.18.

[19] Coupled model intercomparison project (cmip). https://pcmdi.llnl.gov/mips/cmip/about-cmip.html. Accessed: 22.05.18.

[20] deegree homepage. https://www.deegree.org/. Accessed: 19.05.18.

[21] Directory interchange format (dif) writer's guide. https://gcmd.gsfc.nasa.gov/add/difguide/index.html. Accessed: 14.05.18.

[22] Dublin core metadata element set, version 1.1: Reference description. http://dublincore.org/documents/dces/. Accessed: 14.05.18.

[23] Dublin core metadata initiative specifications. http://dublincore.org/specifications/. Accessed: 24.04.18.

[24] Ferret homepage. http://ferret.pmel.noaa.gov/Ferret/. Accessed: 29.05.18.

[25] Geonetwork csw service. https://geonetwork-opensource.org/manuals/2.10.4/eng/developer/xml_services/csw_services.html. Accessed: 14.05.18.

[26] Geoserver homepage. http://geoserver.org/. Accessed: 29.05.18.

[27] Informal interviewing. http://www.qualres.org/HomeInfo-3631.html. Accessed: 19.05.18.

[28] Installing ckan from package. http://docs.ckan.org/en/ckan-2.7.0/maintaining/installing/install-from-package.html. Accessed: 17.04.18.

[29] Leaflet homepage. https://leafletjs.com/. Accessed: 28.05.18.

[30] lxml - xml and html with python. http://lxml.de/. Accessed: 24.04.18.

[31] lxml bug report: " elements cannot be pickled". https://bugs.launchpad.net/lxml/+bug/736708. Accessed: 14.05.18.

[32] Metadata definition. https://www.thefreedictionary.com/metadata. Accessed: 24.04.18.

[33] Nmdc data catalog. http://metadata.nmdc.no/UserInterface/#/. Accessed: 22.05.18.

[34] Norstore research data archive. https://archive.norstore.no/. Accessed: 22.05.18.

[35] Ogc wps 2.0.2 interface standard corrigendum 2. http://docs.opengeospatial.org/is/14-065/14-065.html. Accessed: 19.05.18.

[36] Open archives initiative. https://www.openarchives.org/. Accessed: 27.05.18.

[37] Open archives initiative organization - mission statement. https://www.openarchives.org/organization/. Accessed: 27.05.18.

[38] Open archives initiative protocol for metadata harvesting homepage. https://www.openarchives.org/pmh/. Accessed: 14.05.18.

[39] Open source gis history. http://wiki.osgeo.org/wiki/Open_Source_GIS_History. Accessed: 15.05.18.

[40] Openlayers homepage. https://openlayers.org/. Accessed: 28.05.18.

[41] pip 10.0.01 documentation. https://pip.pypa.io/en/stable/. Accessed: 23.04.18.

[42] Plugin interfaces reference. http://docs.ckan.org/en/ckan-2.7.3/extensions/plugin-interfaces.html. Accessed: 25.04.18.

[43] Plugin interfaces reference. http://docs.ckan.org/en/latest/extensions/plugin-interfaces.html#ckan.plugins.interfaces.IResourceView. Accessed: 12.04.18.

[44] Plugins toolkit reference. http://docs.ckan.org/en/latest/extensions/plugins-toolkit.html. Accessed: 19.04.18.

[45] Postgresql homepage. https://www.postgresql.org/. Accessed: 29.05.18.

[46] Professor howard taylor fisher: Short biography of a pioneer. https://web.archive.org/web/20071213234339/http://www.gis.dce.harvard.edu/fisher/HTFisher.htm. Accessed: 24.04.18.

[47] Pub/sub - redis documentation. https://redis.io/topics/pubsub. Accessed: 07.05.18.

[48] pyoai github. https://github.com/infrae/pyoai. Accessed: 20.04.18.

[49] Quickstart - opendap. https://opendap.github.io/documentation/QuickStart.html. Accessed: 15.05.18.

[50] Rabbitmq homepage. https://www.rabbitmq.com/. Accessed: 28.05.18.

[51] Redis homepage. https://redis.io/. Accessed: 28.05.18.

[52] Solr features. https://lucene.apache.org/solr/features.html. Accessed: 18.04.18.

[53] Solr homepage. https://lucene.apache.org/solr/. Accessed: 18.04.18.

[54] Spatialite homepage. https://www.gaia-gis.it/fossil/libspatialite/index. Accessed: 29.05.18.

[55] Sqlite homepage. https://sqlite.org/index.html. Accessed: 29.05.18.

[56] Stack overflow post: "how to fix lxml assertion error". https://stackoverflow.com/questions/29570715/how-to-fix-lxml-assertion-error. Accessed: 14.05.18.

[57] Stack overflow post: "lxml assertionerror: invalid element proxy". https://stackoverflow.com/questions/31028087/lxml-assertionerror-invalid-element-proxy. Accessed: 14.05.18.

[58] Supervisor: A process control system. http://supervisord.org/. Accessed: 09.05.18.

[59] Thredds catalogs. https://www.unidata.ucar.edu/software/thredds/current/tds/catalog/index.html. Accessed: 09.05.18.

[60] thredds_crawler commit 87c561d. https://github.com/ioos/thredds_crawler/commit/87c561d99edf2725e596a83caba8a400d5461ef7. Accessed: 08.05.18.

[61] thredds_crawler github. https://github.com/ioos/thredds_crawler. Accessed: 02.05.18.

[62] thredds_crawler issue: "cannot retrive dataset metadata". https://github.com/ioos/thredds_crawler/issues/23. Accessed: 14.05.18.

[63] Understanding xml namespaces. https://msdn.microsoft.com/en-us/library/aa468565.aspx. Accessed: 09.05.18.

[64] Using dublin core. http://dublincore.org/documents/usageguide/. Accessed: 14.05.18.

[65] Vagrant homepage. https://www.vagrantup.com/. Accessed: 17.04.18.

[66] Vim - the ubiquitous text editor. https://www.vim.org/. Accessed: 08.05.18.

[67] Virtual environments and packages. `https://docs.python.org/3/tutorial/venv.html`. Accessed: 20.04.18.

[68] What is postgis? `http://postgis.org/`. Accessed: 12.04.18.

[69] Writing extensions tutorial. `http://docs.ckan.org/en/ckan-2.7.3/extensions/tutorial.html`. Accessed: 25.04.18.

[70] The xml c parser and toolkit of gnome. `http://xmlsoft.org/`. Accessed: 24.04.18.

[71] The xslt c library for gnome. `http://xmlsoft.org/XSLT/`. Accessed: 24.04.18.

[72] Thredds data server 4.6. `https://www.unidata.ucar.edu/software/thredds/current/tds/TDS.html`, 2015. Accessed: 10.04.18.

[73] Ricardo Carvalho Amorim, João Aguiar Castro, João Rocha da Silva, and Cristina Ribeiro. A comparison of research data management platforms: architecture, flexible metadata and interoperability. *Universal Access in the Information Society*, 16(4):851–862, Nov 2017.

[74] Ahmad Assaf, Raphaël Troncy, and Aline Senart. Hdl-towards a harmonized dataset model for open data portals. In *USEWOD-PROFILES@ ESWC*, pages 62–74, 2015.

[75] Peter Baumann. Ogc wcs 2.0 interface standard—core. *Open Geospatial Consortium: Wayland, MA, USA*, 2010.

[76] Keith Bennett, Paul Layzell, David Budgen, Pearl Brereton, Linda Macaulay, and Malcolm Munro. Service-based software: the future for flexible software. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 214–221. IEEE, 2000.

[77] Wayne Booth. *The craft of research.* University of Chicago Press, Chicago, 2008.

[78] Kang-Tsung Chang. *Geographic Information System.* John Wiley & Sons, Ltd, 2016.

[79] Open Geospatial Consortium. Catalogue service homepage. `http://www.opengeospatial.org/standards/cat`. Accessed: 24.04.18.

[80] Jeff de La Beaujardiere. Opengis® web map server implementation specification. *Open Geospatial Consortium Inc., OGC*, pages 06–042, 2006.

[81] Yucong Duan, Yuan Cao, and Xiaobing Sun. Various "aas" of everything as a service. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, pages 1–6. IEEE, 2015.

[82] Thomas R Eisenmann, Eric Ries, and Sarah Dillard. Hypothesis-driven entrepreneurship: The lean startup. 2012.

[83] Thore Fechner and Christian Kray. Georeferenced open data and augmented interactive geo-visualizations as catalysts for citizen engagement. *JeDEM-eJournal of eDemocracy and Open Government*, 6(1):14–35, 2014.

[84] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns. 1995. *Reading, Massachusetts: Addison-W esley. ISBN 0-201-63361-2.*

[85] T. Capers Jones. *Estimating Software Costs.* McGraw-Hill, Inc., New York, NY, USA, 2 edition, 2007.

[86] Tom Kralidis. pycsw, metadata publishing just got easier. `http://pycsw.org/`. Accessed: 2018-04-23.

[87] Carl Lagoze and Herbert Van de Sompel. The open archives initiative: Building a low-barrier interoperability framework. In *Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries*, pages 54–62. ACM, 2001.

[88] Markus Lupp. *Open Geospatial Consortium*, pages 815–815. Springer US, Boston, MA, 2008.

[89] NASA. Directory interchange format (dif) standard, August 2017. Accessed 10.04.18.

[90] Douglas Nebert, Arliss Whiteside, and P Vretanos. Opengis catalogue services specification. *Implementation Specification*, 2007.

[91] openresearchdata. openresearchdata/ckanext-oaipmh: Oai-pmh harvester for ckan. `https://github.com/openresearchdata/ckanext-oaipmh`, 2016. Accessed: 24.04.18.

[92] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[93] Robert Scholz, Nikolay Tcholtchev, Philipp Lämmel, and Ina Schieferdecker. A ckan plugin for data harvesting to the hadoop distributed file system. 2017.

[94] Stefan Steiniger and Andrew JS Hunter. Free and open source gis software for building a spatial data infrastructure. *Geospatial free and open source software in the 21st century*, pages 247–261, 2012.

[95] Jianzhi Tang, Yingchao Ren, Chongjun Yang, Lei Shen, and Jun Jiang. A webgis for sharing and integration of multi-source heterogeneous spatial data. In *Geoscience and Remote Sensing Symposium (IGARSS), 2011 IEEE International*, pages 2943–2946. IEEE, 2011.

[96] Roger F. Tomlinson. A geographic information system for regional planning. 1968.

[97] Fubao Zhu, Jinmei Yang, and Qianqian Guo. Emergency gis system based on gml and multi-source spatial data. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 86–90. IEEE, 2011.