FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Marek Čermák

Extraction and representation of unified metadata from files and file systems based on data formats

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Jakub Klímek, Ph.D.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2023

I would like to thank and express my gratitude to my supervisor, RNDr. Jakub Klímek, Ph.D., for his continuous support, guidance, and feedback on this thesis and my adventures in RDF and Linked Data in general.

I am also very thankful to all my friends, colleagues, family and close ones, for their support, patience, ideas, and tolerance when having to discuss this whole endeavour with me.

Title: Extraction and representation of unified metadata from files and file systems based on data formats

Author: Marek Čermák

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Klímek, Ph.D., Department of Software Engineering

Abstract: This thesis documents the process of analyzing, designing, and implementing a software tool able to accept files in various formats, inspect them in depth, and produce a graph in the Resource Description Framework that represents their metadata. Such a description may be useful to any person or system capable of understanding RDF, to provide insight into large sets of files or archives, to allow searching using SPARQL based on concrete domain criteria, or to identify common or distinct entities across different datasets. The results of this thesis may be used by any individual or organization wishing to process files in a semantic and extensible way, to offer users of file hosting sites a wide range of search options, to provide analysts a way to work with metadata in a compact and detailed form, detached from the original source, or to improve systems for processing files with greater control over what kind of data is accepted and processed.

Keywords: RDF, file formats, file format analysis, media, metadata, information extraction

Contents

Introduction

Throughout the history of computers, from the age of the mainframes and continuing its existence into the modern age of the web, one technology has been always present and used by almost everyone – files. Whether they are stored on a floppy disk or accessible through the Internet, computer files are the primary means of recording and labelling digital data of any kind, able to be created, moved, shared, or deleted.

The biggest advantage of files is also their greatest downside – the majority of file systems do not offer any means of truthfully assigning what format a particular file is in, or what its metadata are. If there is an image file containing a photograph, for example, the file system has no means of being certain about it, and it cannot even read its dimensions or find out when the photo was taken[1]. All it sees is just a plain sequence of bytes, oblivious to the media object they may represent.

It is not possible to "fix" file systems and eliminate this issue at this point in the history of their existence, and although there have been attempts in the past to create a new type of media-based file system, they were not successful[2]. It may also be argued that there is no need to fix anything, as this way of storing and understanding files is the best to accommodate for the immense number and variety of file formats in existence, and using any such file system would prevent people from using unsupported file formats efficiently.

There is, however, a better solution to this problem – annotating files using external tools that produce their description, in a suitable format, separate from the data. This has several advantages:

- The file system does not have to maintain any supported file formats; files are still primarily sequences of bytes, and the knowledge of certain formats by the annotation tool only affects the level of detail of their description.
- Choosing a standardized or commonly used format for the description opens up the possibility of using other tools in the respective ecosystem to perform tasks such as exploration, research, or processing of the information within the files through such a description format, without having to be limited to tools that work only with files.
- Additionally, hosting the description separately from the original file makes it possible to operate on datasets consisting of large files without having to download them, using the description first to select those files that are relevant.
- The description can also be used for other purposes, such as for verification, identification of files across multiple datasets, or validation of input data.

The purpose of this thesis is to provide such a tool.

[1]The file extension only serves as a suggestion in what program the file should be opened by the operating system, which also attempts to read such metadata on its own when the format is recognized. This, however, depends on the capabilities of the operating system and cannot be guaranteed on all platforms.

[2]One of them was WinFS, later described in section 2.2.2.

Goals

The goal of this thesis is to design and provide an application that is capable of generating descriptions of input files, by identifying their formats and extracting the relevant metadata. For the purpose of representing the annotations, the Resource Description Framework[3] [1] was chosen. This makes it possible to integrate the output of the software into existing systems working with RDF or Linked Data, and to use solutions like SPARQL [2] or SHACL [3] for further processing.

In practice, such a software could be used for these tasks:

Exploration of file systems

The primary way of using the software is to analyze collections of files and extract useful information. This includes the number of files, their sizes, dates of modification, and the formats of the stored data and their metadata.

As an example, the user may want to filter a large collection of photos and select those that were taken in a specific location, or on a specific date. The software can extract this information based on EXIF metadata stored in the images, and performing a SPARQL query would identify the relevant files.

A similar, more technical example might be exploring a collection of executables and determining their manufacturer, version, or whether they are cryptographically signed.

The output of the software can also be easily accessed as text in formats such as Turtle [4], without having to view it in applications that accept RDF data.

File input validation

A file hosting site may restrict user-uploaded data to a specific form, for example by limiting the range of allowed formats to archive-friendly or non-executable ones, or by requiring the presence of specific files in archives.

The output of this software may be automatically processed using standardized technologies such as OWL [5] or SHACL [3], to determine validity based on the supplied criteria. This is especially useful thanks to the software's ability to recursively traverse through nested file systems, such as archives.

Linking files from different data sources

Another way of using the results obtained via the software is by comparing them to other sources of data or data listings. As an example, there are sites hosting file metadata, including hashes, but not the files themselves. The user may want to find files in their system based on known metadata, such as their hashes, and use the software to determine whether the file system contains occurrences of files that match the criteria.

This could be useful for specific peer-to-peer file sharing services, such as BitTorrent, to be able to determine whether a part of a user's collection of files matches the hashes of files in other collections.

[3]Introduced in detail in section 1.3, its use being justified later in section 4.2.

Extraction of relevant files based on specific criteria

The software's ability to describe files in RDF also makes it possible to use SPARQL to extract individual pieces of data based on conditions expressed in the query. Coupled with the ability to explore even files embedded in archives or executables, the software may be used as a powerful tool to automatically identify and extract data matching a supplied query.

Structure

The following chapters in this thesis are structured as follows:

1. *Preliminaries* – introduction to file formats and other technologies relevant for this work;
2. *Related works* – other pieces of software or systems with a similar focus as this thesis;
3. *Analysis* – identifying the target audience, requirements, and use cases of the software;
4. *Design* – detailed description of the process and its parts, the choice of RDF vocabularies, and the architecture of the software;
5. *Implementation* – the concrete properties of the implementation, the execution environment, libraries, and common interfaces;
6. *Documentation* – documentation of the software for users, web administrators, programmers, and extenders;
7. *Tests* – the overview of the tests that were used to ensure the functionality of the software;
8. *Evaluation* – the summary of the performance of the software in achieving the goals and meeting the requirements.

Conclusion is the final chapter of this thesis, which summarizes the process of building this software and further areas of work.

1. Preliminaries

1.1 Introduction to various file formats

Storing data of various nature in files on a drive or available on the web requires the use of a particular file format. The file format generally specifies the kind of data that it is used to encode, what is the internal structure of the encoded file (i.e. the layout of the bytes or bits composing the raw data), and what algorithms are used to transform the original entity to its encoded form and back.

There is a wide variety of file formats that can be encountered during everyday work with files, both proprietary and standardized. The file format of a particular file is not stored alongside its contents, therefore it is generally not possible to determine the file format used for every conceivable file. However, it is possible to distinguish some file formats based on the differing structure of the bytes, provided that the program can recognize it, or on other telling signs, such as the initial few fixed bytes that many formats use to make the result somewhat self-descriptive (commonly referred to as the "signature" or "magic number"). If the operating system has means to open a file of a particular format, it usually recognizes it based on externally assigned information, such as the file name extension on Windows and Linux or the Uniform Type System code on Apple systems.

1.1.1 Classification

Traditionally, all existing file formats are classified into two categories: text formats and binary formats.

Text formats

The necessity to distinguish text formats has arisen from the differences between major operating systems in relation to the structure of text files: On Unix systems, text files consist of a list of lines, each terminated by the line feed (LF) character (including the last line). Prior to the macOS, the older Apple systems used the carriage return (CR) character to terminate lines. Windows inherits the MS-DOS ending of the carriage return, followed by the line feed character (CRLF), and the last line does not have to be terminated in any way.

The differences in the overall format of text files meant that transferring them between systems required conversion of line endings or other adaptation, in order for the file to be readable. Modern operating systems can usually display non-conforming files properly, as is the necessity due to the infeasibility of transformations in all situations when transferring files over the Internet.

Another important property of text files is the character encoding that determines how the sequence of bytes stored in a text file should map to the sequence of characters that should be displayed, printed, or parsed. This information is usually also not stored alongside the file, therefore the software that opens the file might have to guess the encoding based on the system's default or the frequency of certain bytes in the file.

In modern times, one of the several encodings of the Unicode [6] character set is usually used for new files, but the ability to recognize and process other encodings is still important when working with older sources of files.

Text files can usually be recognized by the absence of specific control code characters except for line separators, as their presence could affect the reading or displaying software in unexpected ways.

Text files are further classified into specific text formats with varying degrees of structure. The completely unstructured text format is called plain text, which is intended to be displayed and processed as-is. More structured formats include those for formatting human-readable text, such as Markdown[1], RTF[2] or TeX[3]. Data conforming to a user-defined format could be stored in formats such as CSV [7] or INI[4]. The end of the spectrum is formed by source codes for various computer languages, conforming to strict specification and unambiguous interpretation.

Binary formats

Binary files consist of a sequence of bytes that should not be interpreted and displayed as characters, and have to use binary mode[5] when encoding. There are no restrictions on the content of a binary file, apart from those imposed by its format.

There are no theoretical restrictions that would prevent choosing a text format instead of a binary format when deciding how to store specific entities, but binary formats are generally preferable when the primary content does not consist of characters or when compression is necessary. Therefore, binary formats are usually selected for storing digitized audiovisual data (such as images, audio, and video), as well as general-purpose file archives containing hierarchies of files and directories. Binary file formats are sometimes also selected when they are intended to be processed by a particular application and not by the user directly, as it is generally harder to edit binary files manually.

The sole knowledge of whether a given format stores a particular kind of media, such as an image or audio, is usually not enough on its own to decode the data, but it may be useful for selecting the optimal transfer speed or compression.

Derived formats

File formats in general may be based on multiple other formats while still logically stored in a single file. In some cases, the base formats themselves are extensible in some way, as is the case for XML [8], which is the underlying format for

[1]`https://daringfireball.net/projects/markdown/`

[2]`https://interoperability.blob.core.windows.net/files/Archive_References/%5bMSFT-RTF%5d.pdf`

[3]`https://tug.org/index.html`

[4]The JSON and XML formats are also used for this purpose, and they are perfectly displayable as text files due to their respective specifications, but they have their own rules for the character encoding and line endings, and thus it is better to think of them as binary formats.

[5]In C, opening a file with `fopen` allows specifying text or binary mode. In binary mode, the LF character is not transformed to the system-appropriate line ending. See `https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/fopen-wfopen?view=msvc-170`.

many other formats such as XHTML [9], SVG [10], SOAP [11], SSML [12], or RDF/XML [13]. In other cases, a common framework could be used for a range of similar formats, such as the OLE compound file[6] used for the original Microsoft Office formats. When the logical structure of a format is close to a file system, archive formats like ZIP[7] may also be used in this case, as is done by Java Archive[8] and Office Open XML[9], used by the newer Microsoft Office formats.

External formats

There are several formats that have requirements on data beyond the bytes of a single file. The most common cases are formats that have to know the name of the file storing the data in order to function properly. This could also be the case of packing formats like gzip [14] or SZDD[10], where the name of the file is necessary to know the proper name of the unpacked file, but the data is recoverable in any case.

Formats that require multiple files range from formats where the coupling is only optional, such as between video and subtitle files, to files where data may be lost when separated, such as multiple archives storing a split file, or CUE/BIN pairs where both files are necessary.

It is also possible for a file to describe an external entity, as is the case of `DESKTOP.INI` files[11] on Windows which provide metadata about the directory that contains them.

1.1.2 Identification and registries

Although for the end user, the format may be hidden and irrelevant, the operating system, browser, or other file processors need a way of knowing how to open a file they encounter. They could attempt to guess the format based on several telltale signs of its contents, but if that fails, there would be no way to inform the user what application to install to recognize and open the file.

There are many ways to provide this information, several of which are worth mentioning.

Name extension

The most common way of specifying the file format is to store it in the filename extension, after the “`.`” character. In older file systems, the extension was a fixed part of the name and was limited to 3 characters; hence most formats commonly used today still follow that pattern.

[6]`https://learn.microsoft.com/en-us/cpp/mfc/containers-compound-files?view=msvc-170`

[7]`https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT`

[8]`https://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html`

[9]`https://www.ecma-international.org/publications-and-standards/standards/ecma-376/`

[10]`https://www.nationalarchives.gov.uk/PRONOM/Format/proFormatSearch.aspx?status=detailReport&id=1249&strPageToDisplay=summary`

[11]`https://learn.microsoft.com/en-us/windows/win32/shell/how-to-customize-folders-with-desktop-ini`

There is no general repository for filename extensions, thus the pairing between the extension and its handler is only determined by the system. In other cases, the extension only informs the user about the purpose of the file, but not about a suitable application to open it, such as `MUS` (music), `VID` (video), `DAT` (data), or `BIN` (binary), all of which usually contain data in a proprietary format specific to the software that uses them.

IANA Media Types

The Internet Assigned Numbers Authority (IANA) maintains a list of registered media types, also known as MIME types[12]. Each format is assigned a code in the form of `type/subtype` which can be used to obtain the registration document, which contains more details about the format [15].

The `type` portion of the Media Type code corresponds to one of the registries and identifies the general type of the data. There are multimedia types such as `image`, `audio`, `video`, `font`, and `model`, component types mostly used in other standards like `multipart` and `message`, and general types `text` and `application`, the former specifying a file that can be viewed as text, and the latter being used for anything that does not fit the other categories.

There are two basic registered types that can be used: `text/plain` for text files and `application/octet-stream` for binary files [16].

The use of Media Types permeates Internet standards, appearing in many protocols such as HTTP [17] or SMTP [18], the `data:` URI scheme [19], and in some metadata formats. Its use solves the ambiguity of extensions, but it is not completely free from issues, mainly due to legacy reasons[13].

Uniform Type Identifiers

On Apple systems, a number of type codes, called Uniform Type Identifiers[14], can be attached to a file that can be recognized by the system. These types form a hierarchy of types residing in different namespaces, following the reverse DNS scheme.

UTIs also link to each other by conformance, allowing to obtain a more general type from a more specific one. There are also type codes that describe other identifiers or schemes.

[12]`https://www.iana.org/assignments/media-types/media-types.xhtml`

[13]Initially, no specific hierarchy was maintained for subtypes, therefore many of them still have inconsistent names. This led to the creation of the `vnd.` (vendor) tree and `prs.` (personal) tree, which indicate the purpose of a format. However, there are still many formats used today that do not follow this pattern. Additionally, while the use of unregistered Media Types is prohibited by the relevant standards, some codes were regularly used despite being unregistered. Their subtypes used the `x-` prefix which earlier standards regarded as being part of the unregistered tree, but the latest version permits their registration if necessary, as the unregistered prefix was redefined to "`x.`". Therefore, one may still encounter confusing situations like an unregistered normal subtype, a registered `x-` subtype, or an unregistered subtype with a different registered name in a specific tree.

[14]`https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/understanding_utis/understand_utis_intro/understand_utis_intro.html`

XML namespaces

The system used to identify semantics of XML elements and attributes does not require the use of a central registry, but still prevents ambiguity and offers some self-description.

XML elements may be defined by a namespace [20], which is identified by a URI [21]. Such a URI technically serves only to disambiguate elements with the same name, but navigating to it should by convention yield a description of the namespace in some form. This form is not mandated by any standard and could only be human-readable, but there has been some practice of hosting the XML Schema [22] description of the format at this location.

PRONOM

There are also repositories whose aim is to identify and preserve various existing formats rather than to be a place for registering new ones. Such a one is PRONOM[15], maintained by the National Archives of the British government. Its database hosts over a thousand file formats, both old and new, including additional metadata such as their extensions, byte order, signatures, or compression types.

Formats identified by PRONOM have a unique identifier, called a PUID (PRONOM Unique Identifier). This identifier is also usable as an `info:` URI [23], for example `info:pronom/fmt/42` identifies the JPEG File Interchange Format 1.00[16].

1.2 Identifying entities on the web

A problem related to identifying and classifying file formats is how to unambiguously identify any sort of resource when communicating with a remote party.

Historically, there have been many different ways to refer to various objects between remote locations, most of which have been replaced or unified by the Uniform Resource Identifier (URI) [21]. Despite that, their understanding is still relevant as they may be used in contemporary formats or encountered in older formats when browsing historical data.

1.2.1 Uniform Resource Identifier

The URI [21] is a form of identifier, nowadays used pervasively throughout the web, being the identifier of choice for many new technologies. The basis of URI was conceived and developed in the early 1990s, then called the Uniform Resource Locator (URL) [24], as a means of encoding a location of a specific resource together with information on how to interpret the location (the scheme). For IP-based protocols, a more complex syntax is usually used:

[15] `https://www.nationalarchives.gov.uk/PRONOM/`

[16] `https://www.nationalarchives.gov.uk/PRONOM/Format/proFormatSearch.aspx?status=detailReport&id=667`

```
<scheme>://[<user>[:<password>]@]<host>[:<port>][/<path>][?<
    ↪query>][#<fragment>]
```

Listing 1.1: The full URI syntax

The fragment, which does not directly participate in retrieving the resource, was not originally specified as a part of a URL, as it is used to refer to a specific part of the resource itself, not determined by any scheme, but solely based on the content type of the resource. Its inclusion in the general syntax is helpful in many situations when the protocol does not permit addressing individual parts of a resource, or when such a mechanism was not provided.

There are numerous registered schemes used to access or identify entities by various means, such as `http:` and `ftp:` for the protocol of the same name [17][25], `file:` for identifying files on a particular machine, regardless of the means of retrieving such a file [26], `mailto:` for sending e-mails, and many others.[17]

Of particular interest are two additional schemes, the `data:` scheme [19] and the `urn:` scheme [28]. The former presents a way of identifying a resource by its data directly in a URI, without the need for any protocol to retrieve it from an external location. The use of such a URI in a protocol or document effectively embeds the resource in it. As there is no inherent limit on the length of a URI, this scheme could be, in theory, used for identifying any piece of data and any part thereof thanks to the fragment.

The `urn:` scheme was created with the primary intention of naming, but not locating, resources. It consists of a hierarchy of namespaces, assigned by the Internet Assigned Numbers Authority, but without any universal mechanism to access the entities to which they refer, provided that a concrete entity could be accessed in the first place.[18]

Internationalized Resource Identifier

The IRI [29] is a technology that extends the URI syntax, capable of storing Unicode characters beyond U+007F directly in its components. The relevant standard also defines the conversion from IRI to URI and vice versa, allowing interoperability with software that does not directly accept IRIs. All other aspects of such identifiers are otherwise identical to those of URIs.

In all further situations in this thesis where URI is mentioned, it is assumed that an IRI could be accepted/produced in its place, unless stated otherwise.

1.2.2 Hash-based identification

Various distributed or peer-to-peer services use identifiers that serve to verify the content to which they were assigned after retrieval, rather than to simply point to its supposed location. There are numerous, mostly non-standard, identifier schemes used in older peer-to-peer clients, and there are also URI schemes designed for this purpose.

[17]An interesting, albeit underused, scheme is the `tag:` scheme [27], not intended for describing the location of existing entities, but for assigning new identifiers unique across space and time to entities.

[18]The meaning of "resource" becomes more abstract when it does not have to be backed by a file, as such could refer to conceptual entities, such as XML namespaces.

An example of such a scheme would be the `magnet:` scheme[19], used for retrieving files based on their characteristics, typically the file hash as a URN derived from the specific hash algorithm, as well as optionally the file length, name, and other information that may assist in retrieving it. The `ni:` scheme [30] works in a similar way, but specifies less metadata about the file, only its content type, and the specification also includes a mapping to a well-known URI scheme [31] in case the authority is included.

1.2.3 Public Identifiers

SGML [32] and XML-based documents could refer to external objects, in the form of document types, embedded entities, or notations, using two types of identifiers: a public identifier and a system identifier [8]. There are no particular restrictions on their structure, but the system identifier is commonly represented by a URI, while the public identifier usually follows the Formal Public Identifier syntax [33].

A Formal Public Identifier consists of several fields that specify the nature of the entity it identifies, including the owner, type of the resource, its description, language, and version. The owner portion is optional and, if present, could specify an owner registered by ISO, an unregistered owner, or the owner of a particular domain name.

Any kind of a public identifier can be converted to a URN in the `urn:public:` namespace [34], using a specific algorithm that preserves the general structure of the identifier.

1.2.4 Universally unique identifiers

A universally unique identifier (UUID) [35] is a 128-bit piece of binary data, generated according to algorithms designed to produce a practically negligible probability of collisions. Such an identifier is usually formatted as a hex-string, and since the structure of it only serves to distinguish it from other UUIDs, there is generally no information in the identifier itself that could lead to its originator or the resource it describes.

Version 1 and 2 UUIDs are generated based on the computer's MAC address and the current precise timestamp, both of which are prone to errors in some cases. Version 3 and 5 UUIDs are the only ones that are based on other identifiers in specific "name spaces", such as DNS or URL, used together with the name string with a hash to form the bits of the UUID. Version 3 uses MD5 [36] as the hash function, while version 5 uses SHA-1 [37]. Version 4 UUIDs are generated solely via a random number generator.

UUIDs can be translated directly into URNs with the `urn:uuid:` prefix.

1.2.5 Object identifiers

Object identifiers (OIDs) [38] form a hierarchy of integer-labelled nodes, together forming a path of dot-separated integers leading from the root node to the specific object. Each node in the hierarchy is given to a specific authority which receives the rights to allocate integers under it.

[19]`https://www.iana.org/assignments/uri-schemes/prov/magnet`

OIDs are very similar to URNs but are generally more efficient to store due to their integer-based nature. On the other hand, the basic OID format does not contain any human-readable labels and must be resolved to find the specific meaning.

OIDs are most commonly used in X.509 certificates [39] to identify various properties and are therefore commonly associated with security.

OIDs are also accessible as URNs with the `urn:oid:` prefix. Additionally, UUIDs may be represented as OIDs in the `2.25` tree[20] as 128-bit integers, or in the Microsoft-hosted `1.2.840.113556.1.8000.2554`[21] as consecutive chunks of the UUID, limiting the maximum length of the individual components.

1.3 Linked data and RDF

A pertinent problem when publishing data is what format to use, as this limits both the nature of the published data and the options the potential consumers of the data will have. An example of such a format is the Resource Description Framework (RDF) [1].

1.3.1 Basics of RDF

Data represented in RDF is, in its basis, a sequence of triples in the form of subject, predicate, object, with restrictions on the type of value in each position. To refer to a particular resource or entity, a URI can be used in all of the 3 positions[22]. Additionally, the object can also denote a literal value that is directly embedded in the triple, which can be a plain string, a language-tagged string, or a value and a URI denoting its type, usually taken from XML Schema [40]. The subject and object positions also allow for a blank node instead of a URI, usable in cases where the URI is unknown or indeterminable.

A collection of these triples can be thought of as an oriented multigraph, with each triple interpreted as a URI-tagged edge between the vertices represented by its subject and object. Semantically, this graph can describe various entities with an unlimited number of properties of arbitrary types, either storing data or pointing to other entities.

As RDF is just a data model, a concrete representation has to be chosen when serializing RDF data into files or over the network. For this purpose, there are various file formats to choose from, RDF/XML [13] and JSON-LD [41] based on existing file formats, or the Turtle language [4], specifically designed for RDF.

Data stores that use RDF can be queried in a standardized way with SPARQL [2] which is designed to match triples in the RDF database based on a specific set of patterns and filters.

[20]`http://oid-info.com/get/2.25`

[21]`http://oid-info.com/get/1.2.840.113556.1.8000.2554`

[22]To be precise, RDF is defined in terms of the IRI instead of URI, and mandates difference between an IRI and a URI formed by percent-encoding the international characters. However, not all pieces of software respect this.

1.3.2 From data to linked data

The RDF data model can be used on its own, but without any agreement on URIs, a data publisher would have to provide not only the actual data, but also the vocabulary alongside it. However, the advantage of using URIs is the possibility of linking to other datasets, where one can navigate and obtain information about the vocabulary used elsewhere. This is accomplished using standard and widely used protocols, such as HTTP.

Equipping vocabularies with rules and relations between different concepts gives rise to ontologies, which can themselves be described using RDF via RDF Schema [42] or the Web Ontology Language (OWL) [5] for describing different parts of information discourse.

There are many vocabularies used in the web of linked data, and anyone is free to define their own. Some commonly used vocabularies are SKOS[23], DCMI Metadata Terms[24], Schema.org[25], or the DBPedia Ontology[26].

[23]`https://www.w3.org/TR/swbp-skos-core-spec/`
[24]`https://www.dublincore.org/specifications/dublin-core/dcmi-terms/`
[25]`https://schema.org/`
[26]`https://dbpedia.org/ontology/`

2. Related works

In this chapter, several other projects related to the topics described in this thesis will be presented. The problem of extracting metadata from files can be naturally separated into two parts, one related to file format analysis and metadata extraction, and one concerning metadata representation in a suitable format. The sections in this chapter reflect this distinction, the first listing individual pieces of software and software suites for file format analysis, while the second describes other methods and schemes for representing metadata.

2.1 File analyzers

The primary purpose of a file analyzer is to look for patterns within a file and to determine the format that corresponds to these patterns. File analyzers vary in terms of additional operations they can perform after the analysis, such as metadata extraction, or whether they can traverse through compound or archive formats. The analysis may be performed either on the whole file, corresponding to a collection of possible formats in which the file could be, or on individual parts of the file, allowing it to identify individual embedded files inside compound formats, data dumps or archives.

2.1.1 PRONOM

PRONOM has already been introduced as a file format registry, but its database can also be used for the purpose of file format classification. The DROID (Digital Record Object Identification)[1] tool utilizes file format signatures stored in the registry to automatically classify files based on their extension or contents. It also supports container formats such as Microsoft Word Document (DOC) files stored in an OLE2 container, or Java Archives stored in a ZIP.

PRONOM's technical database used for classification is stored in a proprietary XML-based format, supporting the matching of binary sequences at various positions in the data, or other factors. While this database could be in principle incorporated into the file format analysis process performed by the software described by this thesis, classification is only one part of file format analysis. The DROID tool is not capable of extraction of file format-specific metadata, hence the other part of the process would have to be handled by other means.

2.1.2 Extractor

The Extractor by Nova Software[2] is a tool for analyzing file formats, allowing not only file format recognition, but also inspection of arbitrary binary files, looking for patterns corresponding to formats of embedded files. The primary purpose of the tool is to extract the embedded files that it finds, but it can also display

[1]`https://cdn.nationalarchives.gov.uk/documents/information-management/droid-user-guide.pdf`

[2]`http://web.archive.org/web/20080725014248/http://www.volny.cz/nova-software/`

metadata for a limited number of formats, such as the dimensions of images or sample rate of audio files.

Extractor also supports plugins via a C or Delphi API which can add additional supported individual formats or compound formats, called “groupfiles”.

2.2 Metadata representation schemes

This section lists several projects that have introduced particular formats or schemes for encoding data or metadata, beyond plain bytes.

2.2.1 NEPOMUK

NEPOMUK[3] is a project containing a series of specifications and a framework which together compose a solution for the Social Semantic Desktop, an evolution of the standard computer desktop with the goal of interconnecting applications and desktops to provide a way for users to express their ideas as semantic information, to be accepted, integrated, and annotated across different applications.

A part of NEPOMUK particularly of interest to this thesis is the NEPOMUK File Ontology[4], an RDF Schema-based ontology for expressing information about various concepts related to files and file formats, such as audiovisual content, documents, archives, executables, fonts, file systems, and their properties.

Although this ontology covers many situations, it lacks detail in others, such as describing arbitrary byte sequences.

2.2.2 WinFS

WinFS[5] was a discontinued project developed by Microsoft, akin to a file system, to store data with a varying degree of structure. The goal of the project was to replace commonly used data formats for storing structured data with records stored in a relational database adhering to a particular schema. “Files” in such a system are not stored as traditional binary blobs, but are exposed as .NET objects having properties specific to their type, such as images providing their dimensions, or documents having a property to store their author, backed by the database. Data in this system does not have to be parsed, as applications can instead acquire information from media objects directly by accessing the properties of the relevant objects.

This project shares some of its goals with this thesis, in particular enabling “semantic search”, a way to search through files based on criteria relevant to their interpretations, rather than just a few attributes shared by all files, such as name, creation date, etc. The presuppositions are different, however, as WinFS expects the user to create the files in a semantic way from the beginning, while the software described in this thesis is made to work with already existing data, without any need to convert it to a semantic file system first.

[3]`https://nepomuk.semanticdesktop.org/`

[4]`https://www.semanticdesktop.org/ontologies/2007/03/22/nfo/v1.2/`

[5]`https://web.archive.org/web/20060602151743/http://msdn.microsoft.com/data/ref/winfs/`

2.3 SPARQL Anything and Facade-X

The SPARQL Anything project[6] [43] provides a SPARQL extension for Apache Jena, as well as a stand-alone application, with the aim of querying arbitrary data using SPARQL. To achieve this possibility, structured data is first re-engineered into the Facade-X meta-model [44], stripping away any domain-specific information. The result of this process is an RDF graph which can be matched using standard SPARQL graph patterns and further processed in the query.

Some of the end goals of this project align with the software presented in this thesis, such as being able to process arbitrary files as RDF data and, as a consequence, "SPARQL anything", however the methodology and design of this process differ from the one presented here, as well as the primary focus: SPARQL Anything focuses on data and its raw structure, expressing it through their domain-independent meta-model, while this work focuses on meta-data, using highly domain-specific ontologies that semantically describe the input data, but not fully encode it. As such, these two approaches can easily co-exist and complement each other.

[6]`https://sparql-anything.cc/`

3. Analysis

This chapter focuses on general aspects of the problem of creating a software to inspect data, analyze file formats and collect metadata.

3.1 Target audience

The people who would benefit from this software can be separated into three main groups, based on common characteristics:

File collections maintainers

These are the people or companies who maintain collections that at some level consist of files, for the purpose of sharing or preservation. The use of this software would improve the quality of their services, including search capabilities or publishing metadata in a standardized form.

Data analysts

These individuals work with specific collections of data and are interested in obtaining information that is relevant to their research. They are expected to understand RDF and related technologies like SPARQL. This software would make it possible for them to extract useful information or files themselves by taking advantage of SPARQL instead of proprietary solutions or manual inspection.

Common users

They usually do not have prior experience working extensively with files, file formats, or RDF. While this software does not offer any means of presenting its results to these users, other tools may be used to visualize the results, or to build SPARQL queries, thanks to the open and standardized nature of the technologies used. The presentation of the RDF output is, however, not part of this thesis.

Even users without technical knowledge can use this software for their own or the society's benefit, if they have access to large data sources. In this case, the output of this software, usually smaller by a few degrees of magnitude than the original data, can be shared with other individuals for offline analysis without requiring the original user to share the actual files.

Regardless of their presence in one of these groups, all users may also vary according to their general programming skill. It is not necessary to have any programming knowledge to use this software, but developers may benefit from their skills by being able to extend the capabilities of the software beyond the default supported components[1].

[1]A developer with this specialization is an extender. Documentation how to extend the software is provided later in section 6.3.

3.2 Requirements

This section describes the requirements the potential users of the software have on its behaviour, based on the expectations of people interested in similar tools, which were gathered from informal discussions and debates.

The requirements stated here can also be evaluated by testing the software according to the scenarios presented in section 7.5.

3.2.1 Functional requirements

The following sections list the functional requirements on the specific features of the software, denoted by *FR* and numbered. There are no specific user roles, so these requirements apply to all of them.

FR1 – File analysis

The user should be able to provide a file or piece of data to the software for analysis as input, and it should be able to determine its properties and produce its description. This can be further broken down into the following:

- FR1.1 – Common file properties

  The software should read the properties of the file in the file system, at the minimum its name, size, and date of last modification, if available.

- FR1.2 – Recognizing known formats

  The software should contain a set of *known formats* and be able to identify files matching one or multiple of the known formats. The set of known formats should contain at least the following[2]:

  - Portable Document Format (PDF, `application/pdf`[3]),
  - Microsoft Portable Executable (EXE, `application/vnd.microsoft.portable-executable`[4]),
  - Extensible Markup Language (XML, `application/xml`[5]),
  - ZIP archives (ZIP, `application/zip`[6]),
  - Ogg audio (OGG, `application/ogg`[7]),
  - WAVE audio (WAV, `audio/vnd.wave`[8]),
  - JPEG JFIF (JPG, `image/jpeg`[9]),

[2]The formats are identified using the IANA Media Types, as described in section 1.1.2.
[3]`https://www.iana.org/assignments/media-types/application/pdf`
[4]`https://www.iana.org/assignments/media-types/application/vnd.microsoft.portable-executable`
[5]`https://www.iana.org/assignments/media-types/application/xml`
[6]`https://www.iana.org/assignments/media-types/application/zip`
[7]`https://www.iana.org/assignments/media-types/application/ogg`
[8]`https://www.iana.org/assignments/wave-avi-codec-registry/wave-avi-codec-registry.xhtml`
[9]`https://www.w3.org/Graphics/JPEG/`

  - Portable Network Graphics (PNG, `image/png`[10]).

- FR1.3 – Properties of known formats

  For each known format, the software should store its MIME type and file name extension. The way a format is matched is implementation-defined[11].

- FR1.4 – Analyzing recognized files

  For every recognized format of an individual file, the software should read the object stored in the file under the specific format, and collect its properties or attributes during the process. The set of browsed properties or attributes is implementation defined.

- FR1.5 – Analyzing unrecognized binary files

  For every binary file that is not matched by any of the known formats, the software should include the signature bytes of the file in the description, if there are any[12].

- FR1.6 – Analyzing unrecognized textual files

  For every textual file that is not matched by any of the known formats, the software should include the first line of the file in the description[13].

- FR1.7 – Computing hashes

  The software should maintain a collection of hash algorithms and use them to compute hashes/digests of the encountered files or data and include them in the description. At least SHA-1 and SHA-256 [45] should be supported.

- FR1.8 – Traversing nested file systems

  A file system consists of directories and files within, but individual files may themselves be serializations of other file systems. The software should be able to identify nested file systems and follow them. As an example, archives stored in other archives should be traversed recursively.

- FR1.9 – Using SPARQL for search

  The user should be able to provide a SPARQL query to the software to be used during the analysis to extract information from the generated description and present it as a result of the query. The way of inputting and processing the query is defined by the software and should be documented.

[10]`https://www.iana.org/assignments/media-types/image/png`

[11]It is not prescribed how a format is matched, as in general, the method is specified by each format individually. A format could be matched using a signature or other byte patterns, but it could also be matched using an arbitrary algorithm or by calling an external API function to perform the matching or parsing the file.

[12]This may give hints to the users of the software or the consumers of its output about the probable format of the file, as many binary formats have a unique signature.

[13]This may be useful to the users of the software in determining the nature of the file, as many text-based formats can be identified by the first line.

- FR1.10 – Using SPARQL for extraction

  The user should be able to provide a SPARQL query to the software to be used during the analysis to select individual sub-files, such as those in archives, and extract them to an output directory, based on a match between the filters or graph patterns in the query and the generated description. The specific form of the query is defined by the software and should be documented.

- FR1.11 – Disabling the detection of specific formats

  The user should be able to remove formats from the set of known formats so that they will not be taken into account during the analysis.

- FR1.12 – Disabling the analysis of specific formats

  The user should be able to skip the analysis of particular formats, so that even if they are detected, no format-specific properties will be collected and they will not be analyzed in detail.

- FR1.13 – Configuring individual components in analysis

  The user should be allowed to control the specific settings that affect the analysis, for example in situations where the result may rely on arbitrary values or guesswork. These settings are defined by every component and should be documented.

FR2 – Description in RDF

The software should express the description of the files obtained during the analysis in RDF, with these additional requirements on its structure:

- FR2.1 – Reuse of existing ontologies

  The software should re-use existing ontologies and vocabularies to express the properties of the analyzed data. The amount of custom-made vocabulary shall be kept at minimum.

- FR2.2 – Use of hierarchical URIs

  The hierarchies present in the analyzed data should be reflected in the URIs for items in those hierarchies, so that it should be possible to determine their position in the hierarchy just by comparing the URIs[14].

[14]For example, if a particular archive is identified using the UUID-based URI `urn:uuid:342842d6-b6f8-40f9-b0d3-6d621b5938e6`, a directory at path `/dir/` in the archive could be identified using `urn:uuid:342842d6-b6f8-40f9-b0d3-6d621b5938e6#/dir/`, and a file named `file.txt` in this directory could be identified using `urn:uuid:342842d6-b6f8-40f9-b0d3-6d621b5938e6#/dir/file.txt`. Note that these URIs are not *relative* to each other in the sense used in RFC 3986 [21], since relative references only operate on the path component of a URI, not its fragment. Not all URI schemes, however, permit hierarchies in the path, as is the case for `urn:uuid:`, so using the fragment may sometimes be the only option.

- FR2.3 – Elimination of blank nodes

  The user should be able to disallow the use of blank nodes in the description, using a URI for every individual instead[15].

- FR2.4 – Separate nodes for the file, its content, and the media object serialized in it

  In the description, a node representing a file should be distinct from the node for its plain content, and both are distinct from the node for the media object actually serialized in the data[16].

FR3 – Presenting the output

The RDF output from the process should be produced and presented to the user, with these additional requirements:

- FR3.1 – Support for common serialization formats

  Usual RDF serialization formats, such as Turtle, RDF/XML, or JSON-LD, should be supported as RDF data storage options.

- FR3.2 – Outputting triples in real-time

  It should be possible to write out the RDF triples directly at the point they are created, as opposed to buffering them in a temporary graph and saving it to the output file at once at the end of the process, if the particular output RDF format supports it[17].

- FR3.3 – Support for formatted output

  The user should be able to enable formatting of the output, to improve readability in an implementation-defined fashion, such as by including whitespace, if the output RDF format supports it[18].

FR4 – Extending the software using plugins

The user should be able to extend the software by providing plugins in a specific format that can add new capabilities while maintaining the previous requirements. Additional requirements are as follows:

- FR4.1 – Support for new formats

  It should be possible to add support for previously unknown formats, allowing the application to detect them.

[15]It is not required that the URI be stable, as it may not be possible to find a stable method of identification of every entity. Conversely, the URIs may also contain random elements.

[16]For example, an image stored in a binary file located in a file system should be represented by 3 nodes in total – one with the file attributes, such as the file name or modification date, one with properties of the data, such as the length or the actual bytes, and one with the properties of the image, like its width or height.

[17]This should be supported by at least Turtle and RDF/XML.

[18]In Turtle and RDF/XML, this should at a minimum add indentation to the output.

- FR4.2 – Support for new analyzers

  It should be possible to modify the analysis process by adding new components to analyze data, or by modifying the existing components and adding new capabilities.

- FR4.3 – Support for new hash algorithms

  It should be possible to provide additional hash algorithms that can be used to compute digests of data, for identification or description.

3.2.2 Non-functional requirements

The following sections specify the expectations about the software or its quality in general, denoted by *NR* and numbered.

NR1 – Execution and portability

The software should be able to be executed in various environments and on various platforms, which should not impair its core functionality. Differences in availability of specific components are permitted, however, as the implementation may not be available for the concrete platform.

- NR1.1 – Running in console

  It should be possible to launch the software from the desktop environment as a console application accepting input from the command line.

- NR1.2 – Running as a web application

  It should be possible to operate the software as a web application, using it solely from the browser.

NR2 – Separation of projects

The software should be separated into multiple projects that each have a clearly defined focus, so that it is possible to use only a part of it without referencing or including all of them. For example, code using the Windows API functions should be limited to one project, so that it is possible to deploy the software to non-Windows platforms by excluding that project without concerns about which other code may be affected.

NR3 – Documentation

The software and its parts should be concisely documented.

- NR3.1 – Use and deployment instructions

  Documentation about building or deploying the software should be provided, as well as documentation for using it, for both the console-based implementation and web-based implementation.

- NR3.2 – Programmer documentation

  Documentation of the code and its parts should be provided, including documentation comments inside the code itself.

- NR3.3 – Extender documentation

  Documentation on how to extend the functionality of the software by adding new components and including them without rebuilding the application should be provided.

NR4 – Tests

The software should include tests that allow assessing its functionality.

- NR4.1 – Unit tests

  Unit tests should be provided for the integral or commonly used parts of the software, showing and verifying their functionality.

- NR4.2 – Complex tests

  More complex tests for the software or its parts as a whole should be provided, testing and evaluating their behaviour.

NR5 – Demo and examples

An easily available demo of the software shall be provided, including examples on how to operate and use it for common purposes.

3.3 Use cases

This section shows possible use cases for the users of the software, covering the functional requirements in section 3.2.1. Since the software does not have multiple user roles, the same actor is implied for all the following use cases – the user. It is also assumed that the software is properly installed or deployed and can be launched by the user.

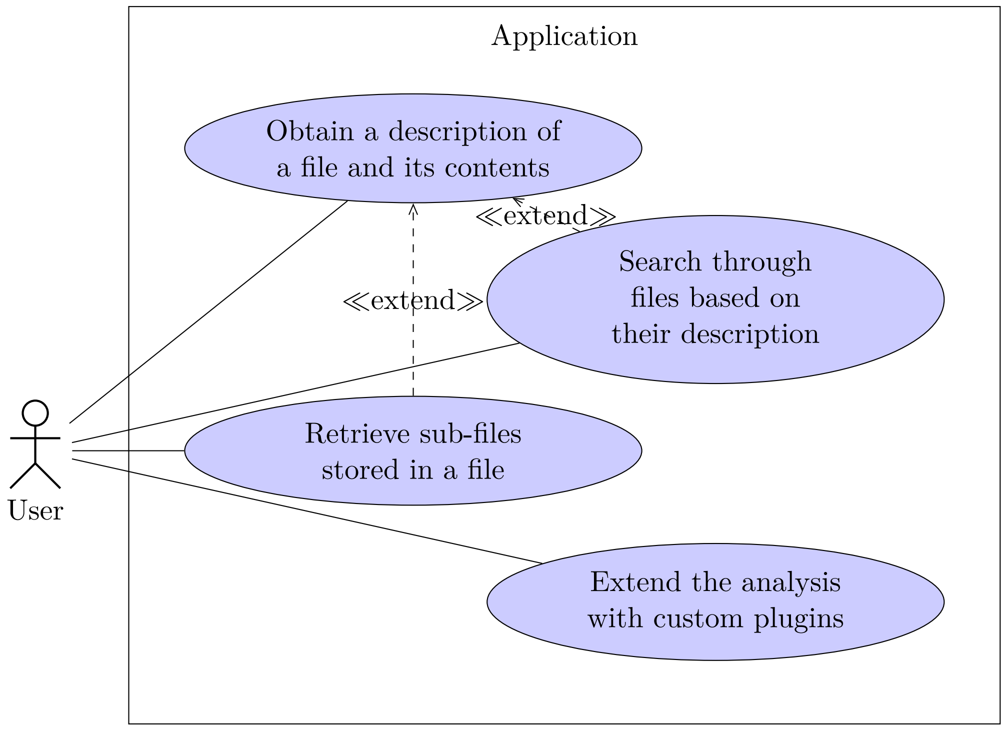

Application
Obtain a description of a file and its contents
≪extend≫
Search through files based on their description
≪extend≫
Retrieve sub-files stored in a file
User
Extend the analysis with custom plugins

Figure 3.1: Use case UML diagram

3.3.1 Obtaining a description of a file and its contents

Description

The user wants to use the software to obtain a full description of a single file, including its properties and the properties of its data and metadata, which can be used by the user for the purpose of search, information retrieval, interlinking, archival, etc.

The output of this process is a detailed RDF description of the file which can be used to derive information from it without having to access the file itself.

Preconditions

- The file to be described is present and accessible in the user's file system.

Steps

1. The user launches the software with the parameters to analyze a file, as specified in the documentation, including the path to the file and a path where to save the output.

2. The software locates the file and reads its common properties, such as the name, date of creation and modification, and size.

3. The software opens the file for reading and derives additional information from its contents, such as whether it is binary or textual, and computes its hashes.

4. The software identifies which of the known formats match the contents of the file.
5. For every matched known format, the software parses the file in the format and analyzes the resulting media object, collecting its properties.
6. The software links the information collected at all these 3 levels, and produces an RDF graph containing all of the collected properties.
7. The graph is serialized to the output file in the appropriate format.

Postconditions

- The status of the progress and result of the analysis are displayed to the user.
- The output file is created and contains information that reflects the analyzed file.

Alternative paths

- The input file may not be found at the specified path. In this case, an error message is displayed and no output file is produced.
- The file may become unavailable, unreadable, or otherwise corrupted during the analysis. In such a situation, an error message is displayed, but the output file is created with the information that the software managed to collect up to this point.
- The media object may itself contain a collection of files. In this case, these files are recursively processed in the same way as the original input file from step 3, and their description is linked to the media object.
- The user may configure the software to suit their needs by removing specific known file formats, hash algorithms, or other components, or by configuring their properties to values different from the default. In such a case, the output of the software is affected in the way specific to the components.

3.3.2 Searching through files based on their description

Description

The user is interested in extracting information represented by a file or its content, based on particular criteria, such as obtaining the properties of specific media objects or looking for the presence of given formats. The software can accept a SPARQL query and evaluate it on the generated description, retrieving the requested information in the result of the query.

Preconditions

- The file to extract from is present and accessible on the user's file system.
- One or multiple SPARQL queries are prepared and accessible through the file system.

Steps

1. The user launches the software with the path to the target file and the paths to the SPARQL queries.
2. The software loads the SPARQL queries.
3. The software opens the input file and follows the same steps as in section 3.3.1 to analyze it and its contents.
4. The loaded SPARQL queries are evaluated on the generated description, and their results are collected.

Postconditions

- The output file contains the results of the evaluation of the SPARQL queries in the provided format.

Alternative paths

- The input file or SPARQL queries may not be found at the specified paths. In such a situation, an error message is displayed, and the software exits.
- The SRARQL queries may not be parseable, may contain syntax errors, or may be otherwise unsuitable to use for extraction. In such a case, an error message is displayed with the error reason, and the process stops.

3.3.3 Retrieving sub-files stored in a file

Description

The user is interested in retrieving actual pieces of data stored inside a file, matching particular criteria, such as resources in an executable file or files in an archive. The software can identify such sub-files by means of a SPARQL query and extract them.

Preconditions

- The file to extract from is present and accessible in the user's file system.
- One or multiple SPARQL queries are prepared and accessible through the file system.

Steps

1. The user launches the software with the path to the target file and the paths to the SPARQL queries.
2. The software loads the SPARQL queries.
3. The software opens the input file and follows the same steps as in section 3.3.1 to analyze it and its contents.
4. When a sub-file that matches any of the queries is encountered, it is copied to an output directory based on the information from the query.

Postconditions

- The list of extracted files is displayed to the user.
- The output directory contains all sub-files matched by any of the queries.

Alternative paths

- The input file or SPARQL queries may not be found at the specified paths. In such a situation, an error message is displayed, and the software exits.
- The SRARQL queries may not be parseable, may contain syntax errors, or may be otherwise unsuitable to use for extraction. In such a case, an error message is displayed with the error reason, and the process stops.

3.3.4 Extending the analysis with custom plugins

Description

The user wants to extend the capabilities of the software to add support for additional known formats, hash algorithms, or other components beyond what the base installation of the software offers. The software accepts plugins that the user can provide for this purpose.

Preconditions

- A plugin file is prepared, either created by the user or obtained from another source.

Steps

1. The user places the plugin file in its appropriate location, as specified in the extender documentation, and launches the software with parameters according to the desired task.
2. The software locates the plugin file and loads it.
3. All components in the file, such as additional known formats or hash algorithms, are added to their respective collections.

4. The process continues as specified by the parameters, but with the components included in it.

Postconditions

- The list of loaded plugins is displayed to the user.
- The components in the plugin are incorporated into the process and affect it when encountered.

Alternative paths

- The plugins may have an invalid format or be otherwise unloadable. In this situation, an error message is displayed to the user, with the name of the erroring plugin.

4. Design

This chapter focuses on the creation of the application from the ground up, the decisions that were made alongside the process, the solutions to the requirements stated during the analysis, and the reasoning behind them.

The design process is divided into four parts, starting with the different methods for analyzing the format of a file, described in section 4.1, followed by the methodology of producing descriptions of files and data in section 4.2. In section 4.3, it is shown what uses such a description should have, and section 4.4 introduces the overall architecture of the software that achieves these goals.

4.1 File format analysis

An integral part of loading any useful data from a file is determining its format, which can be achieved via various methods.

4.1.1 Extension-based

A fast but imprecise method is to only consider the file extension, without inspecting the file's content at all. This is what most operating systems use when opening a file, but for the purposes of this software, this method cannot be used alone, as it relies on the presence and correctness of the extension, which is usually not possible to guarantee[1].

For this reason, the extension can be used merely as a suggestion but not as a determining factor in the whole process. Specifically, a file that uses an extension commonly used for a particular file format must not be classified as having that file format without first using other methods to make sure that the assessment is correct.

4.1.2 Signature-based

A better and often sufficient method is to look for specific sequences of bytes inside the file, known as magic bytes, or the signature if placed at the beginning of the file. Many of file formats use this location to store a short string of ASCII characters with the primary purpose of rejecting invalid or damaged data when the signature does not match, but it can also be compared against a list of known signatures to quickly find out which format it corresponds to, such as `MZ` for Windows and DOS executables, `MThd` for MIDI files, or `PK` for ZIP archives.

This approach is sufficient in many cases, but there are situations when the signature is not present at all, such as in a tar archive or in an MP3 file, at a different position like in a WAV file or MZ-based executables, or the format is determined in a completely different way, such as by using namespaces in XML or metadata files in ZIP-based formats.

These issues must be solved on a per-format basis, such as by using specific heuristics for tar or MP3, looking at multiple positions in the file, or resorting to

[1]In most file systems, a file can be renamed together with its extension.

a more general format like XML and ZIP first, parsing it and then deciding on the more specific format.

4.1.3 Parsing-based

Although the previous approaches are quite efficient, they are not completely precise, as they rely only on heuristics and patterns in existing files, but it is always possible to devise a completely different format that looks similar enough to be confused with the original one. Therefore, the best and most accurate method is to simply attempt to parse the input file in any format not ruled out by the previous checks. The result of the parsing, if successfully obtained, is also required in the following step in order to be described, and thus checking the signature, etc. only serves as an optimization.

4.1.4 Format conflicts

There are cases where a file could be recognized as having multiple formats. One of the reasons for that is some relation between the formats, like WAV being a variant of RIFF, or a Java archive being stored in a ZIP archive. In these cases, the formats would better be expressed in a hierarchy with the most general format, such as RIFF or ZIP, at the top. In other cases, the two formats might be unrelated but could share enough similar structure to allow misinterpretation.

Regardless of whether the reason is accidental or purposeful, it is not a significant complication, provided that multiple formats can be easily described in the output without causing any conflicts.

4.1.5 Text files

All existing text formats are directly or indirectly based on the plain text format, and being able to accurately determine properties of a text file is important to correctly interpret the data within.

Recognizing plain text

Even determining whether a particular file stores text or not is non-trivial, due to the variety of different forms text files were stored as.

Usually, a set of non-text characters is used to determine if a file is binary or not; the presence of such characters indicates that it is most likely not text. However, to accommodate for different encodings and file terminators, only the NUL character is looked for in the file by this software. If it is found at the beginning, or in the middle followed by non-NUL characters, the file is treated as binary straight away, meaning that text files are permitted to end on a variable number of NUL characters. This is useful to permit, as some files or resource storage systems use NUL as a file terminator, which may be found when the file is read.

Analyzing plain text

Once a text file is encountered, there are many properties of it that pose the risk of misinterpretation, such as what sequence of characters is used to delimit lines, whether the file is terminated by some special character and, most importantly, its encoding.

Although some text formats, such as HTML [46], are able to specify the encoding inside the file itself, and some transfer protocols like HTTP [17] can indicate it during the transfer, a text file on disk in most cases does not specify its encoding in any way. Misinterpreting the encoding of a text file leads to an effect known as *mojibake*, when the binary values of characters in the original encoding are interpreted in another encoding, usually resulting in garbled text. In order to avoid this effect in situations where the encoding is not stated explicitly, the encoding has to be guessed via heuristics, for example by computing statistical data about the occurrences of sequences of characters and matching it to profiles of known encodings.

If this process fails or the file contains characters invalid in the guessed encoding, it is better to treat the file as binary, but, similarly to a file being recognized as having multiple formats, a text file could also be recognized as having multiple encodings, and while a garbled text is not usually the desired result, it is also not strictly incorrect.

4.2 Describing file systems

This section covers the problems of representing the information stored in a file system in a structured non-proprietary form that can cover data obtained at any level in its hierarchy.

4.2.1 Choosing the data model and encoding

There are three possibilities for the type of the data model that can be used to represent the metadata: relational, tree-based, and graph-based.

In a relational model, entities are represented as relations grouping together their attributes, usually along with a primary key attribute which is necessary to distinguish distinct but equivalent entities. Typically, data conforming to a relational model is stored in a relational database in fixed tables that conform to a specific schema. This model can be used to represent a relation for directories and a relation for files in a directory, but it is not suitable for natural representation of heterogeneous data that may be found inside the files themselves, since adding a new attribute specific to a single format would require changing the schema of the table.

A tree-based model has a single root, containing or linking to its children, which also link to their children, recursively. The children of a node can usually be distinguished if they have a different relation to the parent. This model could be encoded in JSON or XML, which may or may not specify a schema. Individual file systems naturally map into a tree-based model due to their hierarchical nature, but there are still exceptions. A file system could contain symbolic or hard links to other parts of it, references to entities inside or outside the file system, and entities existing at different levels in a file system may also potentially exist on their own, e.g. a specific image with concrete underlying bytes can be described without first locating it in an actual file system. To prepare for this possibility, a document conforming to this model would have to be split into a collection of files, directories, and data and media objects, each given an identifier in order to be linked from other places, in essence reverting back to the relational approach.

A graph-based model is a perfect match for this case, whereby individual nodes exist on their own and may link to or be linked from an arbitrary number of other nodes. Data in this model can be accessed by locating any of the nodes in the graph. Traversing along its vertices will lead to other parts of the file system.

This makes RDF [1] a suitable framework for representing and encoding any information derived from a file system, as it is capable of identifying any arbitrary entity and properties and links thereof.

4.2.2 Assigning identifiers to file system objects

There are two ways in RDF to refer to resources: by using a URI or through a blank node. While using blank nodes in itself does not limit the ability of the application to formulate facts about the resources it encounters, blank nodes are inherently unstable, and their identity is limited to the document they are defined in. Loading any such document into a triple store prevents the possibility of referring to the individual nodes in a stable manner, unless the triple store supports advanced techniques like using hashing to stabilize blank nodes.

Another reason for preferring to use URIs is the fact that the nature of file systems is hierarchical, and this hierarchy can be easily projected onto a URI structure, making it possible to derive the identifier for any object even without having to traverse the RDF graph.

As a consequence, the use of URIs will be preferred to blank nodes whenever possible. What remains is to describe how to assign URIs to any resources that might be encountered in a file system. It should be noted that the following methods are by no means the only way to achieve this goal, but they were devised with as little arbitrariness in mind as possible, relying on open specifications or common conventions where possible, to increase the chance of interlinking with other possible serialization schemes.

Contrary to RDF recommendations [47], the use of the HTTP(S) scheme in the URIs constructed here is not always possible, simply due to the fact that most resources do not have a primary HTTP representation, and trying to create one would only increase arbitrariness at the risk of breaking dereferenceability when such a service is discontinued.

Data and media objects

The root of the URI hierarchy is formed by representations of contents of individual files, independent of their original locations. To make a distinction between the actual file entity in a file system and its content as a sequence of bytes, the term "data object" is used for the latter.

To identify a data object, the `data:` URI scheme is used, along with the byte sequence stored in the file. At this point, it is also necessary to decide whether the content should be interpreted as being in a text format or a binary format, either externally or using heuristics, to determine whether the media type used by the URI should be `text/plain` or `application/octet-stream`. In the case of text content, the intended encoding should be determined as well. The choice between text and binary format or the encoding of the text does not affect the actual data in any way, rather it gives indication as to how the data should be initially interpreted.

A media object is created when further mechanisms are used to determine the actual structured type of the data, in which case the resulting URI is formed again but with a different media type. There may also be multiple concurrent media types found in the data, or a hierarchy of increasingly more precise media types[2].

Although this approach could in theory identify the content of any file, too large URIs quickly become impractical or difficult to store, the commonly cited length limit being 1024 [19]. Without relying on a central registry of files, it is possible to use hashes to identify such files, as shown in the next section.

Hashes and checksums

There are two parts to this problem: how to identify a data object based on its hash and how to identify the hash itself. The reason for this separation is the fact that several commonly used hash algorithms have been found to be vulnerable to collision attacks and therefore using them to identify a single piece of data would be improper[3].

If a sufficiently strong hash algorithm is used and it can be reasonably expected that the same identifier using it will never be generated for a different piece of data, the `ni:` URI scheme [30] can be used to refer to a data or media object through its base64-encoded hash, created from one of the algorithms in the Named Information Hash Algorithm Registry [30]. The URI for an empty file would therefore be, using the SHA-256 algorithm, `ni:///sha-256;47DEQpj8HBSa-_TImW-5JCeuQeRkm5NMpJWZG3hSuFU`. The scheme also supports specifying the media type of the file along with the hash, so `ni:///sha-256;47DEQpj8HBSa-_TImW-5JCeuQeRkm5NMpJWZG3hSuFU?ct=text/plain` can be used to refer to the empty plain text file. The choice of the media type here is the same as for the `data:` scheme.

For compatibility with systems that use older, potentially unsafe hash algorithms, it is, however, useful to provide support for other hash algorithms

[2]Examples of concurrent media types are mostly text formats that are not distinct enough from other formats, like XML and HTML which could be mistaken for each other in specific cases. Similarly, a program can be written and interpreted in multiple programming languages at the same time, known as a “polyglot”, as a form of recreational programming or to simplify its execution.

A media format hierarchy could be found in generic document types, for example every XML document could also be interpreted as a specific format to reflect its schema, and in container or archive formats, which may be used to create package formats.

[3]It is, however, possible to reinterpret a possibly ambiguous identifier as “the first file ever created by humanity with this particular hash”, under the assumption that all files having the same hash that followed were created just to prove that the collision is possible and are, in fact, not identifiable by that hash anymore. These semantics may be useful to adopt for a sufficiently strong hash algorithm, but any weak hash or checksum algorithms are better to be interpreted in a different way, as illustrated in the following paragraphs.

or checksums, as there are existing databases that use them to lookup additional metadata. Informally, URN namespaces such as `urn:md5:`, `urn:sha1:` or `urn:crc32:` have been used for this purpose, despite not being registered. When working with resources in this form, it is important to define the precise meaning of the entities denoted by these URIs. In a peer-to-peer system, a URI like `urn:md5:D41D8CD98F00B204E9800998ECF8427E` (the MD5 hash of the empty file) is generally satisfiable with any file that has the same hash, not just the empty file[4], and therefore such a URI should be interpreted in a way that reflects this. When this form of a URI is encountered, its general interpretation is not a specific file with the exact hash, but any such file, as a concept broader than any concrete file that matches the hash.

This approach makes it possible to link a uniquely identified hash to almost any resource, not just files, while being able to handle any degree of uncertainty or loss of information. There are examples of such hashes like the BitTorrent Info Hash used for files or directories and their metadata, or a suite of techniques for difference-based hashing of images known collectively as dHash. Hashing of XML or RDF documents also falls into this category when canonicalization is used.

Files and directories

File system objects are represented as part of a hierarchy with the file system itself at its root, combined with the path to the individual files and directories.

The root of the file system is itself represented by an arbitrary URI, with different possibilities depending on the intended usage. For creating a serialization of an HTTP server, FTP server, or a local file system, the appropriate URI scheme could be used, but the URIs created this way are not guaranteed to be resolvable to the same content on a different machine or at a different time. A safer alternative is to generate a new UUID-based URN every time the file system is browsed, but this makes the result non-deterministic and hard to track changes with. A stable and deterministic option, yet one that may not always be available, is to first store the file system itself in an archive format and use the available methods to generate an identifier for the media object it represents. It is also possible to use a specific format for the archive that does not need to be concretely present as a single file and could instead be formed, hashed, and disposed of on the fly as the file system is browsed.

Due to the large number of sources and possible formats the file system could be provided with, there are no standardized options to choose from to refer to objects within. In case of URI schemes that expose a file system-like hierarchy (i.e. `http(s):`, `ftp:`, and similar), the path can be easily included in its proper location in the URI, but the `ni:` and `data:` schemes do not offer this possibility.

One option is to use the fragment portion of the URI to store the path. However, this is not usually provided as part of the official registration document for media types at the IANA, making the format of the fragment always arbitrary to a degree. Regardless of this, a URI like `ni:///sha-256;...?ct=application/zip#/path/file.txt`, inspired by other hierarchical fragment schemes such as JSON Pointer [48], might be interpreted to refer to a file located

[4]Peer-to-peer clients also check that the length of the file matches after it is downloaded, but this information is not reflected in the hash identifier itself.

at /path/file.txt in the specific ZIP archive. To achieve at least some compatibility with other proprietary fragment formats, the XPointer framework [49] could be adopted, resulting in a URI like ni:///sha-256;...?ct=application/zip#xmlns(fs=http://example.org/fs)fs:path(/path/file.txt). This decreases the risk of collisions with other types of fragments, at the cost of increasing the length of every URI.

Another possibility is to use a specific URI scheme to construct a new URI that conveys the meaning of opening an archive and addressing individual files within it. The only registered URI scheme, at the time of writing, that permits this use, is the jar: scheme, which supports ZIP files, but there are applications that understand other proprietary schemes. In this case, the URI would look like jar:ni:///sha-256;...?ct=application/zip!/path/file.txt.

The following conventions are also adopted for every path, regardless of the encoding scheme: every path starts with /, and a path to the contents of a directory, separate from the directory object itself, ends with /.

Path objects

Although the identifier for an actual concrete file can be derived from the URI of its parent directory and the name of the file, a local naming scheme is usually used when the file system is accessed by the operating system. For example, on a Windows operating system, a path like C:\Windows\notepad.exe uniquely identifies a file in the "Windows" directory on a volume assigned to the letter C.

For the purpose of identifying files in the scope of the system, the file: URI scheme [26] may be used, so, for example, an URI of file:///C:/Windows/notepad.exe could be used to identify the same file. However, this does not work when translated into RDF, as there is no concept of a local system, and even if there were, such a document would change meaning when moved across different systems.

Therefore, under the assumption that all URIs used in an RDF document are stable and unambiguous, an approach similar to that in the case of hashes can be taken. In this interpretation, file:///C:/Windows/notepad.exe does not refer to a single file on the local system but to any file on any file system that can be accessed by that path, simultaneously. Additionally, a virtual file system may be imagined in the C:\Windows directory, allowing for the use of file:///notepad.exe as a broader concept for any file named notepad.exe, regardless of its location.

Although it may be argued that abusing the file: scheme simply to refer to paths, which could be easily represented as data nodes, is not necessary, this approach has some advantages. First, while an RDF graph can be easily navigated by traversing data nodes, many databases only offer the ability to browse and navigate through URI or blank nodes, thus an operation like "find all files that share a name with this file" would require writing a SPARQL query just for this purpose. Second, it may be useful to describe the paths themselves, as they also form a hierarchy and may be associated with broader or narrower concepts.

It will also be necessary to produce two distinct URIs for a single directory; one to identify it as a file in its parent directory and the other other to identify it as a container of other files. The ending / character serves as the distinguishing factor: a URI like file:///C:/Windows identifies the directory as a file node in `file:///C:/`, while `file:///C:/Windows/` identifies it as a collection of files. There is a special case for the root `file:///`, because `file://` is invalid. In this case, `file:///` is given the first meaning of a file node, while `file:///./` is interpreted with the second meaning.

Many paths have an extension, also called a suffix, which determines how to interpret the file located at that path. It is reasonable to also provide a URI for the extension, as this makes it possible to use a database that associates extensions with file formats to guess the format of a file, all just by navigating through its path. The `https://w3id.org/uri4uri/suffix/` namespace is used for this purpose, as there is no designated URI scheme just for extensions.

Lastly, it is a common practice on Windows systems to use environment variables in paths, to refer to specific configurable locations, for example using `%WinDir%\notepad.exe` to prepare for the possibility that the Windows directory may be moved to a different location. At the cost of possible ambiguity, such a "path" may also be encoded as `file:///WINDIR:/notepad.exe`[5], imagining that a file system may be mapped under that specific variable name. However, using variables in other portions of the path remains unsupported, and devising a scheme for this case is beyond the scope of this thesis.

Source objects

Although every data object can be solely identified by its hash, it is usually not sufficient to use such an identifier to locate the file without additional information. In some cases, it is possible to create a URI that combines the source file system and the path of the resource, to be able to locate it without the need to traverse the path in the graph every time. As an example, a URI like `ni:///sha-256;...?ct=application/zip#/path/file.txt` is not enough to retrieve the file if there is no mapping that would resolve the hash of the archive, but if such an archive was downloaded from `https://example.org/archive.zip`, its location could be substituted instead of the `ni:` URI, forming `https://example.org/archive.zip#/path/file.txt`.

It is important to be aware that this association is temporally unstable, as the source URI may become unavailable or start providing different data. The Wayback Machine hosted by the Internet Archive[6] may be used instead to form a stable reference to the resource at a specific time of retrieval, e.g. `https://web.archive.org/web/20220216013606if_/https://example.org/archive.zip`.

[5]Windows environment variables are case-insensitive.

[6]`https://web.archive.org/`

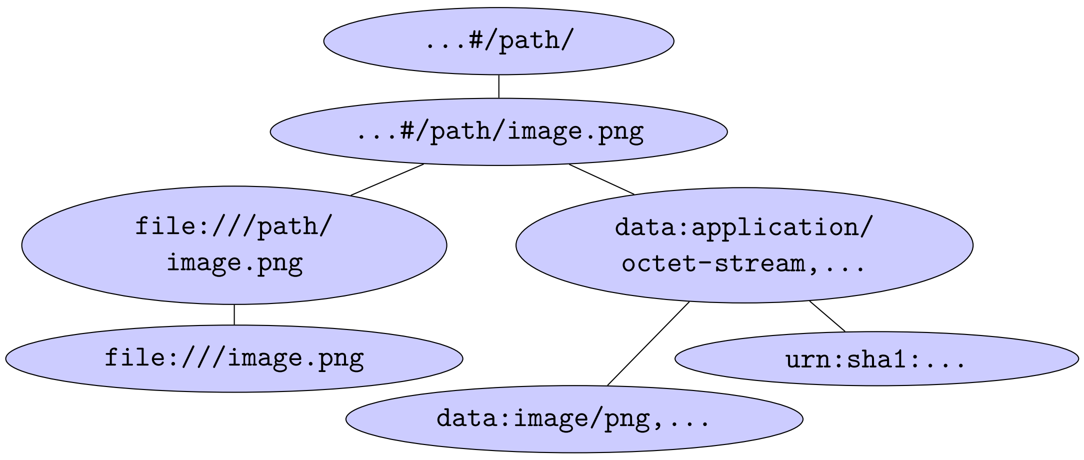


Figure 4.1: Example graph of nodes corresponding to an image in a directory

4.2.3 Choice of vocabularies

Once the relevant classes of entities in the universe of discourse are established, they can serve as the basis for selecting the optimal RDF vocabularies to describe them. While it is possible to build a custom ontology perfectly tailored to the needs of the application, using a combination of already established ontologies increases compatibility with other similar projects. There are several factors that affect the selection:

Prior use. A vocabulary already used in several projects has a greater chance of being recognized and correctly processed by tools designed to work with such projects.

Precision. Ability to describe entities and distinguish concepts to the depth required by the application. As an example, vocabularies might conflate the concept of a file, the raw content within, and its interpretation.

Compatibility. It is better to use vocabularies with reasonably aligned concepts and properties that can be used together.

Reasoning power. Vocabularies backed by well-defined consistent ontologies increase the possibility of deriving additional useful data from the resulting graph.

Based on these criteria, the following vocabularies were chosen:

Nepomuk File Ontology[7]

The Nepomuk File Ontology is part of a larger family of ontologies within the NEPOMUK project. This ontology is intended for use in file system serializations, offering a specialized vocabulary for defining file systems of different types, files and folders, and types of media.

[7]`https://www.semanticdesktop.org/ontologies/2007/03/22/nfo/`

This ontology offers separation between storage and information objects, making it possible to differentiate between a file and the resource encoded therein, but it lacks necessary precision as it defines files as sequences of bytes rather than as storages of such sequences, which is necessary in order to be able to identify the sequences themselves.

Representing Content in RDF [50]

The issue of representing data free of any specific storage is solved by an ontology devised by the W3C Working Group, which refers to this form of data as "content". It offers specific classes and properties for describing textual, binary, and XML content as distinct but linked entities, suitable for combination with the `data:` and `ni:` URI schemes that also support specific content types.

However, this ontology does not offer any means of interlinking various types of content together, or specifying additional metadata.

DCMI Metadata Terms[8]

The Dublin Core Metadata Initiative vocabulary provides a basic set of properties and classes to describe resources in general, regardless of what kind of resources they are.

This is a commonly used ontology, and as such its use is useful when other specialized ontologies do not offer similar capabilities. Specifically, it is recommended by the W3C Working Group to use alongside the Content ontology to link different representations of resources together as formats of each other.

Schema.org[9]

The Schema.org vocabulary is a general-purpose web-oriented vocabulary covering a broad range of subjects relevant when annotating documents representing concrete physical or digital objects. This vocabulary comes into play for the purpose of this application when describing individual media objects and their metadata, alongside the more technically oriented Nepomuk File Ontology. As it is more widely used, its general properties are more suitable to use compared to synonymous properties in the Nepomuk File Ontology.

The Security Vocabulary[10]

This ontology is a work of the Credentials Community Group and covers various types of entities in this area, including certificates, signatures, keys, and digests.

The ability to describe digests on their own including their type and value is very useful, as it is an accurate representation of the associated URNs, however it does not offer any ability to associate a digest with an object it represents.

[8] `https://www.dublincore.org/specifications/dublin-core/dcmi-terms/`
[9] `https://schema.org/`
[10] `https://w3c-ccg.github.io/security-vocab/`

Additional concepts

Regardless of the expressive power of these vocabularies, there are still some parts of the serialization that are not covered by them. These parts of ontology shall be defined on top of the common vocabularies, using as much of the existing terms to describe them.

One such case is linking the resource identifying a hash or digest to a resource that has that particular digest. While there are several vocabularies, such as the Semantic Web Publishing Vocabulary [51] with its `swp:digest` property, they either restrict the domain of such a property, making it unusable for most cases, or require the object to be a literal, like the aforementioned example.

```
at:digest a owl:ObjectProperty, nrl:DefiningProperty ;
  skos:prefLabel "hash"@en ;
  skos:altLabel "digest"@en ;
  dcterms:description "Links a resource to an object describing
     ↪ its hash."@en ;
  rdfs:range sec:Digest ;
  rdfs:subPropertyOf skos:broadMatch, schema:identifier ;
  skos:closeMatch nfo:hasHash, swp:digest, <urn:oid
     ↪:1.2.840.113549.1.7.1.5> .
```

Listing 4.1: The definition of `at:digest`

As defined, every `at:digest` property implies `skos:broadMatch`, which means that the hash object is in some sense “broader” than the object being hashed, as it, as a concept, can be thought of as encompassing all resources with the same particular hash.[11]

Another relation that has to be defined is the relation between a file system object and its path object, or between two different path objects where one ends with the other, or between a path object and its extension object:

```
at:pathObject a owl:ObjectProperty, owl:TransitiveProperty, nrl
   ↪:DefiningProperty ;
  skos:prefLabel "path"@en ;
  skos:altLabel "path object"@en ;
  dcterms:description "Links an entity to a file object
     ↪generalizing its location."@en ;
  rdfs:subPropertyOf skos:broader .

at:extensionObject a owl:ObjectProperty, nrl:DefiningProperty ;
  skos:prefLabel "extension"@en ;
  skos:altLabel "extension object"@en ;
  dcterms:description "Links an entity to an extension object
     ↪indicating its encoding."@en ;
  rdfs:subPropertyOf skos:broader .
```

Listing 4.2: The definitiond of `at:pathObject` and `at:extensionObject`

[11]Specifically, `skos:broadMatch` also implies that the subject concept is in a different “concept scheme” than the object concept. The intended meaning here is that all resources that could be hashed, such as files, directories, or pieces of data, are fundamentally in a different semantic category than all their hashes.

Again, the presence of these properties implies `skos:broader`, meaning that e.g. a path `dir/file.txt` is in some sense broader than an actual file located at that particular location in some file system, and that `file.txt` is even much broader.

4.2.4 Describing a file system

The following sections show how to map concepts in a file system to RDF. For the most part, entities in a file system will be described using the NEPOMUK vocabularies. URIs of the individual entities in a file system are formed in accordance with the principles described in section 4.2.2 and section 4.2.2.

In the following examples, a simple instance of a file system is used, containing one folder with a single file.

```
/
  folder
    file.txt
```

Listing 4.3: Example file system hierarchy

File system root

The file system encompasses all the directories and files within. It may be a representation of a drive or a volume located on a physical machine, or the root of a web server, but the file system may also be serialized in a container format, such as an archive or hard drive image. This is indicated using `rdf:type`, usually pointing to a subclass of `nfo:DataContainer`:

```
# If the file system is stored in an archive:
<> a nfo:Archive .
# Alternatively, if it is stored in an image (CD, DVD etc.):
<> a nfo:FilesystemImage .
# Or if the type is not known or relevant:
<> a nfo:DataContainer .
```

Listing 4.4: Options to describe the file system root

The root of the file system may also be thought of as a unique directory-like entity in the file system, and as such describable using properties like `nfo:fileLastModified`, but also treated as a folder containing all the top-level files and directories. This is realized as two distinct nodes, linked using `nie:interpretedAs` from the Nepomuk Information Element Ontology[12] in a manner described in greater detail in the following sections.

```
<> a nfo:DataContainer ;
# The file system is the root of its own hierarchy.
  at:pathObject <file:///> ;
# Its contents are treated as a folder.
  nie:interpretedAs </> .
```

[12]https://www.semanticdesktop.org/ontologies/2007/01/19/nie/

```
</> a nfo:Folder ;
  skos:prefLabel "/" ;
  at:pathObject <file:///./> .
```

Listing 4.5: Properties of the root file and directory

Files and file system objects

All files, here including special file system objects[13], are represented by `nfo:FileDataObject`, or a specialized class, such as `nfo:ArchiveItem` or `nfo:EmbeddedFileDataObject` for specific kinds of file systems. All files are linked using `at:pathObject` to a broader concept representing their position in the file system hierarchy, as described in section 4.2.2.

```
</folder/file.txt> a nfo:FileDataObject ;
# This file is also an item in an archive.
  a nfo:ArchiveItem ;
  at:pathObject <file:///folder/file.txt> .
  skos:prefLabel "/folder/file.txt" .
```

Listing 4.6: Example of a file node

Most file systems provide a way to represent and preserve useful metadata about files alongside their name and size, such as the creation date, the date of last modification, the revision, and others. Properties such as `nfo:fileName`, `nfo:fileSize`, `nfo:fileCreated`, `nfo:fileLastModified` and similar are used to store these attributes.

```
</folder/file.txt>
  nfo:fileName "file.txt" ;
  nfo:fileLastModified "1999-09-22T05:08:46.000000+02:00"^^xsd:
     ↪dateTime ;
  nfo:fileSize 11 .
```

Listing 4.7: Other properties of a file node

Lastly, a file can be linked to its parent directory using `nfo:belongsToContainer`, and to its content using `nfo:interpretedAs`[14].

```
</folder/file.txt>
  nfo:belongsToContainer </fs:folder/> ;
# The file can be interpreted as text.
  nie:interpretedAs [ a cnt:ContentAsText ] .
```

Listing 4.8: Linking a file to its directory and contents

Directories and folders

Every directory in a file system is actually expressed using two nodes: one when the directory is viewed as an item in a folder, i.e. as an `nfo:FileDataObject`, and one for the actual collection of its contents, as an `nfo:Folder`. These two

[13]Directories, symbolic links, etc.

[14]The content of text or binary bytes is described more in section 4.2.5.

concepts are linked together using `nie:interpretedAs`, which is used to link an instance of `nie:DataObject`[15] to an instance of `nie:InformationElement` it represents. The two resources are also distinguished by the presence of a trailing slash (`/`) at the end of the path.

```
# A directory as a file system object uses the same set of
   ↪applicable properties used for files:
</folder> a nfo:FileDataObject ;
  a nfo:ArchiveItem ;
  at:pathObject <file:///folder> ;
  skos:prefLabel "/folder" ;
  nfo:fileName "folder" ;
  nfo:belongsToContainer </> ;
# but it is not interpreted as data:
  nie:interpretedAs </folder/> .
```

Listing 4.9: Properties of a directory as a file

The actual collection of files is represented as an instance of `nfo:Folder`. Its contents are linked using `nfo:belongsToContainer`, as previously described.

```
</folder/> a nfo:Folder ;
  skos:prefLabel "/folder/" ;
  at:pathObject <file:///folder/> .
```

Listing 4.10: Properties of a directory as a container

4.2.5 Describing content

In an actual semantic system, like in WinFS or the Semantic Desktop, files could store images, documents, etc. "directly", but in most environments, text or binary data is the only kind of data accessible from a file without any additional analysis.

For this reason, the vocabulary specified by Representing Content in RDF [50] is used first to describe the content of a file at the basic level, before any concrete format is found[16].

This vocabulary provides classes for describing different "forms" of content, such as `cnt:ContentAsText` when the content is interpreted as plain text, and `cnt:ContentAsBase64` when the content is treated as raw binary data, expressed in base64.

For text content, the relevant properties are `cnt:chars` storing the text itself as `xsd:string`, and `cnt:characterEncoding` specifying the original encoding[17]. There is no specified property to represent the size of the data, but `dcterms:extent` can be used in combination with a specific datatype, such as `dt:byte` or another DBpedia Datatype[18]:

[15]Files, streams, data items, services etc.

[16]In NEPOMUK alone, this step is skipped, as the `nfo:FileDataObject` instance also hosts properties for its byte content, like `nfo:hasHash` or `nie:byteSize`

[17]The value of `cnt:chars` is always in Unicode, hence the use of `cnt:characterEncoding`. This property can also be used in binary data derived from text, in which case it specifies the encoding.

[18]`http://mappings.dbpedia.org/index.php/DBpedia_Datatypes#InformationUnit`

```
# If possible, the file is identified purely by its content,
  ↪such as using the data: scheme.
<data:,hello%20world> a cnt:ContentAsText ;
  cnt:chars "hello world"^^xsd:string ;
# us-ascii is the default encoding for the data: scheme.
  cnt:characterEncoding "us-ascii" ;
  skos:prefLabel "text (11 B)"@en ;
  dcterms:extent "11"^^dt:byte .
```

Listing 4.11: Example description of a text content with the characters `hello world`

For binary data, only the `cnt:bytes` property is relevant, storing the data itself as `xsd:base64Binary`:

```
<data:application/octet-stream,%01%02> a cnt:ContentAsBase64 ;
  cnt:bytes "AQI="^^xsd:base64Binary ;
  skos:prefLabel "binary data (2 B)"@en ;
  dcterms:extent "2"^^dt:byte .
```

Listing 4.12: Example description of a binary content with bytes `01-02`

Per the recommendations in Representing Content in RDF, `dcterms:hasFormat` should be used to link between different formats or "forms" of the same resource. It may be used to link from the binary representation to the textual representation, or to a media object. For this application, it is not necessary to provide both `cnt:ContentAsBase64` and `cnt:ContentAsText`, as either is suitable to describe the data in a form that can be used to derive the other representation.

4.2.6 Describing hashes

A particular hash or digest, produced by processing the data using a specific algorithm, is usually defined by two pieces of information: the concrete hash algorithm, and its output.

This entity can be described via The Security Vocabulary, using the hash node identified by a URI as described in section 4.2.2. Such a resource has the type `sec:Digest` and uses two properties: `sec:digestAlgorithm` to identify the particular hash algorithm to produce the hash, and `sec:digestValue` to store its value, as base64. Each hash instance is linked to the node representing the hashed object using `at:digest`, a sub-property of `skos:broader`:

```
<data:application/octet-stream,%01%02>
  at:digest <urn:md5:0CB988D042A7F28DD5FE2B55B3F5AC7A> .

<urn:md5:0CB988D042A7F28DD5FE2B55B3F5AC7A>
  a sec:Digest ;
  sec:digestAlgorithm <http://www.w3.org/2001/04/xmldsig-more#
    ↪md5> ;
  sec:digestValue "DLmI0EKn8o3V/itVs/Wseg==" .
```

Listing 4.13: Linking an MD5 hash node to a content

Although hashes are usually computed from binary data, any object may be hashed using a standard binary hash algorithm when serialized or formatted to produce binary data. In some cases, e.g. text, JSON or XML, there may be multiple distinct serialization techniques, differing in features like line endings, encoding, order of properties/attributes, etc. This can be specified using the `sec:canonicalizationAlgorithm` property on the hash resource[19].

4.2.7 Describing media objects

The information obtained during file format analysis can be used to construct the description of the media object actually stored in the file, such as an image, executable, document, etc.

This description largely depends on the nature of the media object. However, all media objects use the `schema:MediaObject` class, and their encoding can be specified using `schema:encodingFormat`:

```
_:container a schema:MediaObject ;
  schema:encodingFormat <https://w3id.org/uri4uri/mime/
    ↪application/zip> ;
  skos:prefLabel "ZIP object (924 B)"@en .
```

Listing 4.14: Example of the base properties of a media object

In the following subsections, descriptions of several media formats or categories of them will be shown.

Images

Most image objects share a number of properties, such as dimensions, resolution, or pixel format. These properties are expressible using the Nepomuk File Ontology, containing `nfo:width`, `nfo:height`, `nfo:colorDepth`, and similar:

```
_:image a schema:MediaObject ;
  a schema:ImageObject ;
  a nfo:Image ;
  nfo:width 266 ;
  nfo:height 132 ;
  nfo:horizontalResolution 95.9866 ;
  nfo:verticalResolution 95.9866 ;
  nfo:colorDepth 24 ;
  schema:encodingFormat <https://w3id.org/uri4uri/mime/image/
    ↪png> ;
  skos:prefLabel "PNG object (266x132, 8-bit)"@en .
```

Listing 4.15: Example image object description

A thumbnail may be linked to the image object, encoded as a `data:` URI:

[19]This usage poses certain issues, however, since the canonicalization algorithm is not an intrinsic property of the hash value, but of the process that led to it. For this reason, the property should not be used in the case of insecure or weak hash or checksum algorithms or generally in situations where it is likely for the same hash to be produced for different and differently canonicalized pieces of data. In that case, a more proper usage would be to encode the canonicalization algorithm in the URI node, or to use a blank node and link to the general hash without canonicalization using `skos:broader`.

```
_:image
  schema:thumbnail <data:image/png;base64,...> .
```

Listing 4.16: Linking a thumbnail to an image object

The pixel data of the image can also be hashed. This is distinct from the hash of the image when stored in a format like PNG and helps to identify images with the same raw pixels but stored in different formats:

```
_:image
  at:digest <urn:md5:5827DCB7E21952923D380E3946777E12> ;
  at:digest <urn:sha1:VS53ZHBDU65NLVBGKJCENV68N96FDVZF> .
```

Listing 4.17: Linking hashes to an image object

Other metadata stored in the image may also be derived using specific vocabularies, such as for EXIF tags[20] stored in JPEG images:

```
_:image
  exif:make "SONY"^^xsd:string ;
  exif:model "DSC-WX350"^^xsd:string ;
  exif:orientation 1 ;
  exif:xResolution 350 ;
  exif:yResolution 350 ;
  exif:resolutionUnit 2 ;
  exif:software "DSC-WX350 v2.00"^^xsd:string ;
  exif:dateTime "2021-02-22T13:01:17.000000"^^xsd:dateTime ;
  exif:yCbCrPositioning 2 .
```

Listing 4.18: EXIF properties on an image object

Sound and music

Similarly to images, audio objects can be primarily described by the Nepomuk File Ontology, with properties such as `nfo:duration`, `nfo:sampleRate`, `nfo:channels`, and others:

```
_:audio a schema:MediaObject ;
  a schema:AudioObject ;
  a nfo:Audio ;
  nfo:channels 1 ;
  nfo:bitsPerSample 8 ;
  nfo:sampleRate "22050"^^dt:hertz ;
  nfo:duration "PT0.0080272S"^^xsd:duration ;
  schema:encodingFormat <https://w3id.org/uri4uri/mime/audio/
    ↪vnd.wave> ;
  skos:prefLabel "WAV object (8-bit, 22050 Hz, 1 channel)"@en .
```

Listing 4.19: Example audio object description

Audio files may also be equipped with metadata sections, such as ID3, commonly used for MP3 files. Nepomuk ID3 Ontology[21] can be used to describe those:

[20] https://www.w3.org/2003/12/exif/
[21] https://www.semanticdesktop.org/ontologies/2007/05/10/nid3/

```
_:audio a nid3:ID3Audio ;
  nid3:title "Adagio for Strings" ;
  nid3:leadArtist "Boston Symphony" ;
  nid3:composer "Samuel Barber" ;
  nid3:albumTitle "Samuel Barber - Adagio, Op.11" ;
  nid3:contentType "Classical" ;
  nid3:recordingYear 1997 ;
  nid3:trackNumber 1 ;
  a schema:AudioObject ;
  a nfo:Audio ;
  schema:encodingFormat <https://w3id.org/uri4uri/mime/audio/
     ↪mpeg> ;
  skos:prefLabel "MP3 object (10.69 MiB)"@en .
```

Listing 4.20: ID3 properties on an audio object

Videos and media containers

Videos can be described, in the simpler cases, as a combination of both image and audio content, using both sets of relevant properties. However, many container formats used to store video also support the storage of multiple streams of visual or audio content. Multiple separate media streams in a single container can be linked using `nfo:hasMediaStream`:

```
_:video a schema:MediaObject ;
  a schema:VideoObject ;
  a nfo:Video ;
  nfo:duration "PT0.04S"^^xsd:duration ;
# Properties of the primary stream:
  nfo:width 1280 ;
  nfo:height 1024 ;
  nfo:hasMediaStream [
# The first video stream:
    a nfo:MediaStream ;
    a nfo:Video ;
    nfo:width 1280 ;
    nfo:height 1024 ;
    skos:prefLabel "0:Video (1280x1024)"
  ] ;
  nfo:hasMediaStream [
# The second video stream:
    a nfo:MediaStream ;
    a nfo:Video ;
    nfo:width 640 ;
    nfo:height 480 ;
    skos:prefLabel "1:Video (640x480)"
  ] .
```

Listing 4.21: Example video object description

Executables

Executables form a category of versatile file formats that usually store executable code, but also metadata such as version number, information about the creator, or copyright. Data files may also be stored alongside the code in an executable in the form of resources.

Most properties usage for storing executable metadata can be taken from Schema.org, such as `schema:softwareVersion` or `schema:copyrightNotice`:

```
_:executable a schema:MediaObject ;
  a schema:SoftwareApplication ;
  a nfo:Executable ;
# For the PE format used by Windows, these values are taken
   ↪from the VS_VERSION_INFO structure:
  schema:name "Object Manager Namespace Viewer" ; #
     ↪InternalName
  dcterms:creator "Sysinternals"@en-us ; # CompanyName
  schema:copyrightNotice "Copyright (c) 1996-2010 Mark
     ↪Russinovich"@en-us ; # LegalCopyright
  schema:softwareVersion "2.22" ; # ProductVersion
  dcterms:description "Sysinternals Winobj"@en-us ; #
     ↪FileDescription
  dbo:originalName "Winobj.exe" . # OriginalFilename
```

Listing 4.22: Example executable description

Other contents of the executable can be represented as embedded files, linked using `nie:hasPart`:

```
_:executable
  nie:hasPart [
    a nfo:FileDataObject ;
    a nfo:EmbeddedFileDataObject
    # other properties of the file
  ] .
```

Listing 4.23: Linking embedded resources to an executable object

Archives

Archive formats such as ZIP or RAR store multiple compressed files, usually organized into directories, forming their own individual file system. For the most part, the resulting description is not different from the description of any other file system, with properties such as `nfo:fileName` or `nfo:belongsToContainer`:

```
_:archive a schema:MediaObject ;
  a nfo:Archive ;
# This forms the root of the file system.
  at:pathObject <file:///> ;
  nie:interpretedAs _:folder .

# It can be interpreted as a directory.
```

```
_:folder a nfo:Folder ;
  skos:prefLabel "/" ;
  at:pathObject <file:///./> .

_:file a nfo:FileDataObject ;
  a nfo:ArchiveItem ;
  at:pathObject <file:///file.txt> ;
  skos:prefLabel "/file.txt" ;
  nfo:fileName "file.txt" ;
  nfo:belongsToContainer _:folder .
```
Listing 4.24: Example archive object description

XML

Documents in XML can be described in part by XML-related classes and properties defined in Representing Content in RDF [50], capable of expressing the non-structural parts of XML, i.e. the XML declaration and document type declaration, while the XML structure can be represented using the RDF schema for the XML Infoset [52]. The descriptions are produced per the instructions in the respective specifications, as an instance of `cnt:ContentAsXML` and `xis:Document`, sharing the same node:

```
_:document a schema:MediaObject ;
# As an XML content, using cnt:
  a cnt:ContentAsXML ;
  cnt:version "1.0" ;
  cnt:standalone "no" ;
  cnt:dtDecl [
    a cnt:DoctypeDecl ;
    cnt:doctypeName "svg" ;
    cnt:publicId "-//W3C//DTD SVG 1.1//EN" ;
# Interlinking with other documents with the same PUBLIC
   ↪identifier:
    rdfs:seeAlso <urn:publicid:-:W3C:DTD+SVG+1.1:EN>
  ] ;
# As an XML document, using xis:
  a xis:Document ;
  xis:documentElement [
    a xis:Element ;
    xis:localName "svg" ;
    xis:name "svg" ;
    xis:namespaceName "http://www.w3.org/2000/svg"^^xsd:anyURI
       ↪;
# Intelinking with other documents using the same root
   ↪namespace:
    rdfs:seeAlso <http://www.w3.org/2000/svg>
  ] ;
  schema:encodingFormat <https://w3id.org/uri4uri/mime/
     ↪application/xml> ;
  skos:prefLabel "XML object (svg)"@en .
```
Listing 4.25: Example XML content description

In most cases, an XML document is in itself only an encoding of a more concrete format. In that case, `dcterms:hasFormat` can be used again to link to a more specific representation of the resource, such as a `schema:ImageObject` for SVG [10] in the example above.

Documents

Digital documents, usually stored in various office formats, commonly store metadata about the publication itself, such as its author, date of creation, keywords, etc., in addition to useful properties of the text, such as the number of words, paragraphs or pages. Most of relevant properties are covered by the DCMI Metadata Terms and Schema.org, such as `dcterms:creator`, `dcterms:title`, `schema:keywords` or `schema:category`, while Nepomuk File Ontology provides properties like `nfo:lineCount` and `nfo:pageCount`. The document resource itself is an instance of `schema:DigitalDocument` or `nfo:Document` or a specialized class such as `nfo:TextDocument` or `nfo:Spreadsheet`.

```
_:document
  a schema:DigitalDocument , nfo:Document ;
  dcterms:creator "Document Author" ;
  dcterms:title "Document Title" ;
  dcterms:created "2012-07-08T13:24:00+02:00"^^xsd:dateTime ;
  schema:version "1" ;
  nfo:characterCount 24 ;
  nfo:wordCount 4 ;
  nfo:pageCount 1 ;
  nfo:lineCount 1 ;
  schema:encodingFormat <https://w3id.org/uri4uri/mime/
    ↪application/msword> ;
  skos:prefLabel "DOC object (Document Title)"@en .
```
Listing 4.26: Example document description

Implied formats

It is always possible that files in an unrecognized format are encountered. Even if so, it may be desirable to derive some information from the contents of such files, to be able to group together files that are likely to be in the same format, to analyze them later, and then link them to the proper format.

Such a format is fully implied by various signs which are, by convention, shared among most formats, making it possible to use these signs to create an identifier for the implied format. The prefix `application/x.` is used here to form an unregistered private MIME type, based on signs derived from the specific categories of formats.[22]

[22]The `x.` prefix is not recommended to be used for public data, but providing a distinct MIME type is necessary to form `data:` and `ni:` URIs.

Binary formats. Most binary files, by convention, start with a sequence of usually 2 or 4 ASCII characters, called a signature. This designation does not come under any authority for registration and can be freely chosen by the designers of the format, but it is usually in their best interest to pick a signature that is distinct from commonly used formats.

For example, if a file starting with the characters `RIFF` in ASCII is encountered, but there is no registered routine matching its format, a default representation could still be produced:

```
_:object a schema:MediaObject ;
  schema:encodingFormat <https://w3id.org/uri4uri/mime/
      ↪application/x.sig.riff> ;
  skos:prefLabel "RIFF object (5.08 KiB)"@en .
```

Listing 4.27: Example media object for the `RIFF` signature-based implied format

After manual analysis, the user may find out that RIFF is a container format commonly used for WAVE audio files. The user may choose to represent this relation via SKOS, linking the more concrete format to the implied one using `skos:broader`[23]:

```
<https://w3id.org/uri4uri/mime/audio/vnd.wave> skos:broader <
    ↪https://w3id.org/uri4uri/mime/application/x.sig.riff> .
```

Listing 4.28: Suggested linking of an implied format and a real format

Text formats. Text files do not have any prevalent convention to indicate the format like binary files, but on Unix systems, it is common to denote scripts with the "shebang" sequence, `#!`, to indicate which interpreter to use when executing the file, in the form `#!/path/to/interpreter arguments`.

```
# Implied from #!/bin/sh
_:object a schema:MediaObject ;
  schema:encodingFormat <https://w3id.org/uri4uri/mime/
      ↪application/x.exec.sh> ;
  skos:prefLabel "SH object (1.12 KiB)"@en .
```

Listing 4.29: Example media object for the `sh` interpreter-based implied format

XML-based formats. Formats based on XML are usually distinguished by the full name of the root element, i.e. its local name and namespace URI[24]:

```
_:xml-object a schema:MediaObject ;
# The namespace URI can be broken down using its hierarchy,
    ↪removing irrelevant characters:
  schema:encodingFormat <https://w3id.org/uri4uri/mime/
      ↪application/x.ns.org.w3.www.http.2000.svg.svg+xml> ;
```

[23]The logic here is that all WAVE files start with "RIFF" as the signature, but not all RIFF files are WAVE, as other formats such as AVI also use RIFF containers, and, in general, another format distinct from RIFF but using "RIFF" as the signature can be encountered.

[24]If the namespace URI is not present and the document has a DTD with a PUBLIC identifier, it can be converted to the `urn:publicid:` URN namespace and used instead of the namespace.

```
  # The format can also be described by the DTD
    schema:encodingFormat <urn:publicid:-:W3C:DTD+SVG+1.1:EN> ;
  # Taken from the root element's local name:
    skos:prefLabel "SVG object"@en .
```

Listing 4.30: Example media object for the {`http://www.w3.org/2000/svg`}`svg` XML name-based implied format

4.2.8 Handling invalid data

When working with arbitrary files, it is possible to encounter malicious, corrupted, or misinterpreted files that could potentially cause the software to corrupt its output or make it unloadable by other software working with RDF.

There are several cases that need special handling throughout the process, which are described in the following sections.

Long URIs

URIs can store any sequence of bytes, thanks to percent-encoding, and the length of this sequence is theoretically unlimited. However, some software may place limits on the length of individual URIs, such as the limit of 1900 bytes in OpenLink Virtuoso[25].

URIs exceeding a pre-configured character limit can be shortened without sacrificing uniqueness by converting to a version 5 UUID, using the following steps as specified by RFC 4122 [35]:

- A byte array is created, starting with the encoding of `NameSpace_URL`[26], followed by the URI encoded in UTF-8.
- An SHA-1 hash is computed, truncated to 16 bytes.
- A version 5 UUID is created from the hash, by setting the appropriate version fields.
- A new URN is formed from the UUID using the `urn:uuid:` prefix.

This process in essence turns a URI identifying a resource into a fixed-size UUID, by convention identifying the same resource, and encoding it as a URN of that resource.

Transforming URIs this way is useful in situations where parts of the parsed input files are used to form URIs. It is also possible to employ this to identify content, transforming it from its `data:` URI, but this is inefficient and impractical compared to using the actual hash of the file with the `ni:` scheme.

It is generally impossible to retrieve back the original URI from its shortened form, however it is possible to create another RDF graph to store such links, using `owl:sameAs`.

[25]`http://docs.openlinksw.com/virtuoso/setshortenlongurisparam/`

[26]Defined as `6ba7b811-9dad-11d1-80b4-00c04fd430c8` by RFC 4122 [35].

Invalid characters in literals

Literals in RDF are more restricted than how strings are usually handled in common programming languages. To facilitate encoding in RDF/XML, the lexical value must not contain any characters not covered by the `Char` production in XML 1.0 [8]. The forbidden characters are all ASCII control characters with the exception of tabs and line breaks, UTF-16 surrogate characters, and the non-characters U+FFFE and U+FFFF.

Additionally, there are some Unicode characters that are valid in XML 1.0, but whose purpose is to be combined with the preceding character, and as such could cause problems when displaying if found at the beginning of literal. These characters include the `M` (Marks) Unicode category [40], containing combining characters, and the zero-width joiner, U+200D, used to link characters which should be displayed together.

It is possible that these literals may inadvertently appear if taken from unsanitized input. When a triple containing such a literal is to be written to the output, one option is to skip the triple altogether. Another option, to preserve the original value, is to encode it in a way that makes it possible to decode it, but also to retrieve the original type of the literal, if specified. For this purpose, the JSON-LD [41] encoding of the literal is used:

```
// A literal containing a single NUL character:
{"@value":"\u0000"}
// With a language tag:
{"@value":"\u0000","@language":"en"}
// With a datatype:
{"@value":"\u0000","@type":"http://example.org/"}
```

Listing 4.31: Example JSON-LD encoding of invalid RDF literals

This literal is used instead of the original one, with the `rdf:JSON` datatype. While this is not semantically equivalent to the original literal and is not actually valid JSON-LD, this form of encoding can be usually easily decoded as most programming languages have support for JSON.

Invalid characters in files

Files recognized as text may often contain invalid characters, especially older documents using various control characters to separate pages or mark the end of the file. While this case is already covered by encoding invalid literals as JSON, the value of the `cnt:chars` property may be encoded in a way that is not as disruptive as JSON.

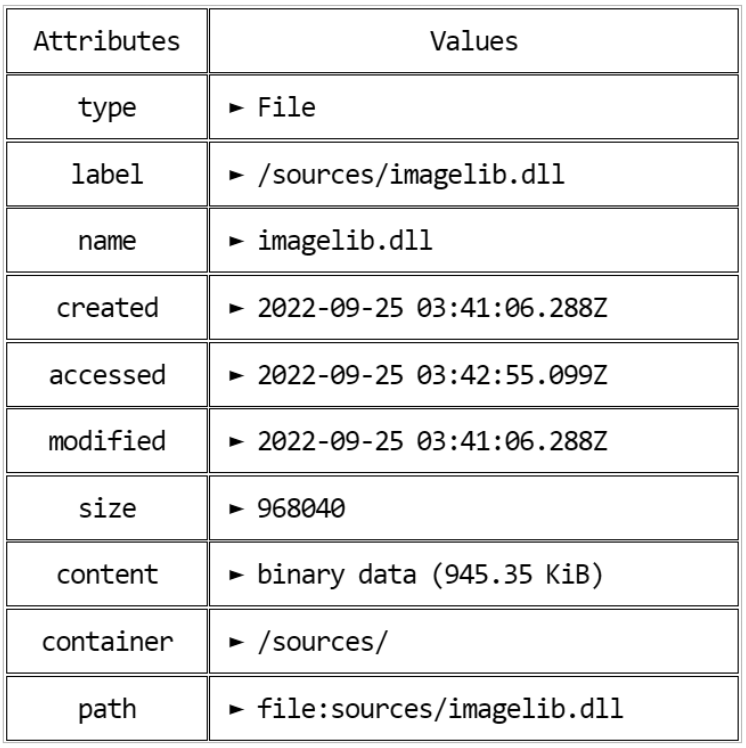
There is a block of Unicode characters starting at U+2400 (␀) used as replacement symbols for the corresponding forbidden control characters. This process is still reversible as long as the original encoding of the file does not support the actual replacement characters, in which case they could be transformed back after checking the `cnt:characterEncoding` property. For this reason, Unicode text files cannot take advantage of this transformation.

4.3 Interaction with output

The output of this process is a gradually formed RDF graph matching and describing the input files and directories, and their contents. It is up to the user to decide how to use the RDF data, but there are several purposes worth presenting.

4.3.1 Storing RDF as file metadata

One possibility, useful for file-hosting sites and archives, is to simply store the RDF output alongside the original files, either as text or uploaded to a triple store. When the file is presented to a viewer of such a site, the RDF metadata can be used to give the viewer an almost full overview of the file and its contents. The data can be used in its original form or processed to make it more presentable to the viewer.



Attributes	Values
type	► File
label	► /sources/imagelib.dll
name	► imagelib.dll
created	► 2022-09-25 03:41:06.288Z
accessed	► 2022-09-25 03:42:55.099Z
modified	► 2022-09-25 03:41:06.288Z
size	► 968040
content	► binary data (945.35 KiB)
container	► /sources/
path	► file:sources/imagelib.dll

Figure 4.2: Example of a potential display of RDF file metadata

If the triple store is equipped with a SPARQL endpoint, it gives the user the possibility to look for specific information about the files or search based on arbitrary criteria.

4.3.2 Semantic file search

As the RDF graph is formed from files at various levels in the file system, SPARQL queries can be used to identify files based on arbitrary criteria expressed via the RDF properties used in the descriptions.

This can be used to select files based on certain properties or to extract them if found in archives.

4.3.3 Validation and processing

The RDF output can be used for automatic validation of arbitrary user input or to derive useful information from it. For example, a SPARQL query can be used to determine whether the input contains files of a specific kind, with particular sizes, or, when coupled with external sources, to look for files whose extension does not match their recognized type.

All these constraints can be combined for complex validation using RDF-based languages such as SHACL [3] or OWL [5].

4.3.4 Extraction of hashes

If enabled, hashes calculated from files are part of the output data, identified by URIs with prefixes such as `urn:md5:`, `urn:sha1:` and similar. These hashes can be used in peer-to-peer file sharing networks to locate the file, for example via a `magnet:` link.

4.4 Architecture

The software uses object-oriented programming to separate the logic and behaviour between entities occurring in the system. The two core abstractions are *linked nodes*, to capture description of RDF resources, and *analyzers*, to transform various entities to linked nodes and describe them.

These objects are handled by the *inspector*, selecting the appropriate analyzer for the specific kind of input. During the analysis, other entities may be discovered or produced in addition to the linked node. These are recursively handled back to the inspector to delegate.

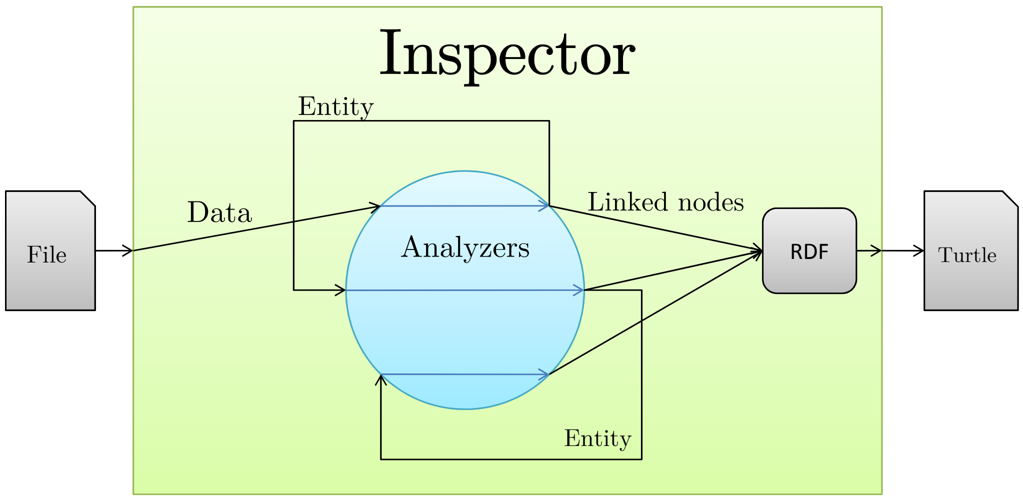


Figure 4.3: Overview of the inspector object and its interactions

The analyzers themselves are distinguished based on the type of input they accept, and may as well contain other components relevant to the analysis of such input:

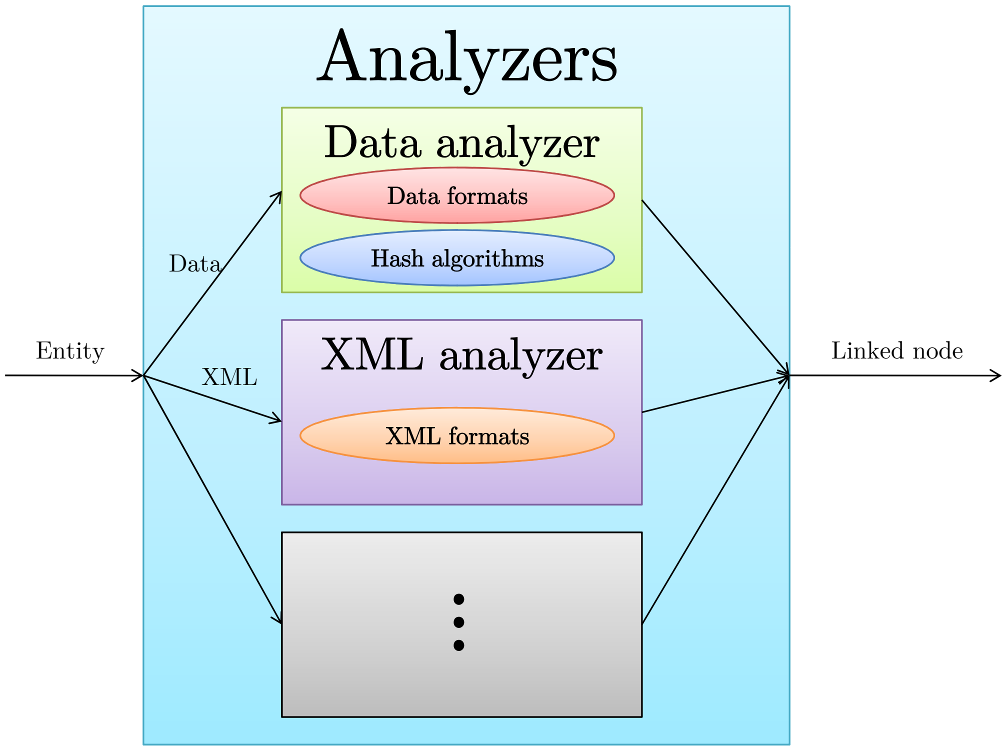


Figure 4.4: Overview of several example analyzers

The abstractions are explained in greater detail in the following sections, while their concrete representation in code is given later in section 5.4.

4.4.1 Linked nodes

A linked node is a write-only abstraction of an RDF node appearing in the subject position in a graph. It can also be thought of as the description of an RDF resource, usually identified by a URI.

This abstraction is intended to be used to unify various methods of producing RDF data, such as direct output to a file or construction of an RDF graph in memory, thus there are no operations which would be able to retrieve information from an RDF graph; instead the available methods only allow setting properties on an RDF resource, equivalent to adding triples to an RDF graph.

Aside from setting its RDF properties, a linked node supports derivation of a new linked node using a sub-name. This operation is used in situations where the node corresponds to a group of entities, such as when identifying a file by its name in a directory containing it or locating a vocabulary item in a vocabulary.

4.4.2 Analyzers

Analyzers are objects whose purpose is to describe entities of specific types in RDF, through an instance of a linked node representing the entity. The linked node may be provided externally, but it may also be created by the analyzer itself, by means of derivation from the parent node, or only based on the received entity.

Analyzers are kept in a sequential collection ordered based on the specificity of individual analyzers; analyzers of most derived types come first, followed by analyzers of less derived types. The inspector starts the analysis process by traversing the collection and looking for the first analyzer supporting the entity. If found, the analyzer receives the entity and a reference to the analyzer collection through its `Analyze` method and returns a linked node representing the entity. From within the analyzer, nested entities can also be identified, separately analyzed, and linked back to the original node. This process is recursively repeated through the hierarchy formed by the entities and visited by the analyzers.

File analyzer

The file analyzer is usually the first analyzer in the process, used for files and directories. Its task is to produce the RDF description based on properties of the file system entities themselves, not their data, such as the name, path, date of last modification, etc.

Some hash algorithms, such as the BitTorrent Info Hash, require a concrete file system object instead of just the data, in order to produce the hash not only from the contents but also from names or other metadata. For this reason, the file analyzer also keeps a collection of file-based hash algorithms which are used on files and directories.

Individual files are handled by the data analyzer as plain binary data.

Data analyzer

The data analyzer examines binary data and determines its properties.

Some properties assigned by the data analyzer are based on heuristics, such as whether the input is binary or textual, or its encoding in the latter case.[27]

The analyzer maintains a collection of data-based hash algorithms, such as MD5 or SHA-1, all of which are used to process data exceeding a certain size. Data below this threshold is identified using the `data:` URI scheme, hence it is not necessary to compute the hashes at all, while larger data results in computing all of the hashes, and possibly picking one of them to form its `ni:` URI.[28]

Another responsibility of the data analyzer is to store a list of supported binary file formats and attempt to recognize them from the data. This process is separated into several steps, outlined here and later described in detail in section 5.4.3:

[27]The factual correctness of the result is not affected by the accuracy of these heuristics, however, as the resulting node can be thought of as representing just one possible, and ultimately valid, interpretation of the data, regardless of its author's intentions.

[28]This threshold can be selected based on an estimate of the size of the RDF output. Having to store all of the hashes would be inefficient for small files, compared to using a `data:` URI.

- *CheckHeader* is called to match the few initial bytes of the data. This method should be lightweight, without initiating any complex parsing process, enough to quickly filter out cases where the format cannot match due to incorrect signature, etc.

- *Match* is called to attempt to parse the data and produce an object describing it. The object is retrieved from within *Match* via a callback because some formats might require cleanup after the object is no longer needed.

- If an object is successfully produced from *Match*, it is wrapped in a *format object* and processed by other analyzers.

- If *Match* ends in an exception before producing a result, or produces no result at all, the format is considered to be unmatched.

All information collected about the object, including whether any formats were recognized, is stored in a *data object* and left to be analyzed further.

Data object analyzer

This analyzer exists to describe the *data object* produced by the data analyzer. It expresses its properties in RDF, and if no explicit formats were detected, it creates the implied format and sends it to analysis.

This analyzer is not strictly necessary, as the RDF properties could be assigned directly in the data analyzer, however this makes it possible to replace the way the RDF properties are assigned while keeping the data and file format analysis the same.

Format object analyzer

A *format object* stores the object produced from file format analysis, together with the object that identifies the format. This analyzer uses the stored format to obtain the MIME type, necessary to produce the URI of the format object, and the extension to form the label. Formats derived from XML also expose the namespace of the root element and the `PUBLIC` and `SYSTEM` identifiers of the document type.

This can be expressed using the following inequality:

$$size_{inline}(l) \leq size_{hashed}(l)$$

$$size_{inline}(l) = \begin{cases} \#\texttt{"data:,"} + T + 2l & \text{(text file)} \\ \#\texttt{"data:;base64,"} + T + 2l\frac{4}{3} & \text{(binary file)} \end{cases}$$

$$size_{hashed}(l) = \#\texttt{"ni:///;?ct="} + \#niname_{primaryhash} + size_{primaryhash}(l) \cdot \frac{4}{3} + \sum_{hash} (3T + urisize_{hash}(l))$$

Where l is the length of the input file and T is the size of a single RDF triple. The fraction $\frac{4}{3}$ is used as an estimate of the increase in size after base64-encoding the data. $urisize_{hash}(l)$ corresponds to the total size of a URI identifying the particular hash, including the URI prefix and the length of the encoded output of the hash function, possibly variable based on the input length.

The object inside the format object is not constrained to any specific type, and is completely opaque to the analyzer. Instead, it is extracted and passed to any analyzer that supports it, but keeping the same linked node.

4.4.3 Formats

A format is, generally, a representation of a specific structure, form, or restriction of a less constrained medium. Binary formats specify the structure of a byte sequence, but other formats may specify the structure of more derived objects, such as an XML format constraining XML documents.

The number of categories of formats is not limited in any way because each individual analyzer may use its own collection of specific formats, similarly to the data analyzer.

Generally, each format exposes a lightweight "check" method that only tests whether the object may be a valid instance of the format, based on common signs, leaving more complex parsing or allocations to the "match" method.

Binary formats

Binary formats specify the structure of plain binary data, a sequence of bytes. The *CheckHeader* method only operates on the file "header", i.e. a smaller initial section of the file, whose maximum length is specified in *HeaderLength*. When reading data, the data analyzer tries to read at least *HeaderLength* + 1 bytes, allowing the format to determine whether there is any data after the header, and potentially reject files consisting of only the header or a smaller section.

If the header check succeeds, *Match* should be called, given the full binary data.

XML formats

Other kinds of formats imposing a particular structure on data, such as XML-based formats, may also be distinguished in a similar manner. Here, the analogy to *CheckHeader* is the *CheckDocument* method, which is given the Document Type Declaration and the information about the root element, such as its name or attributes. If the check succeeds, the *Match* method is called to read the whole XML document, similarly to binary formats.

4.4.4 Hash algorithms

Hash algorithms are, in general, pieces of code that digest a particular object, returning a short sequence of bytes that may be used to distinguish the object among several others. This definition applies to a wide range of algorithms, ranging from weak checksums, like CRC32, to hashes of non-binary data, such as dHash used for images.

Hash algorithms are, like formats, attached to concrete analyzers that have the best access to their required input, for example hash algorithms for processing binary data are used by the data analyzer, hash algorithms requiring file nodes are used by the file analyzer, and so on.

To form URI nodes identifying individual hashes (section 4.2.2), each hash algorithm must store a URI prefix, such as `urn:sha1:`, and an encoding method for the bytes of the hash[29]. In the case the hash can be also used as the primary identifier for the object, forming a `ni:` URI [30], the hash algorithm must also store its `ni` name[30] or its multihash [54] identifier[31]. To describe the hash resource, the hash algorithm should also define its URI[32]. Additionally, in order for the data analyzer to estimate the minimum length of data to hash, the hash algorithm also stores the size of the hash in bytes.

Concrete hash algorithms can be categorized based on the kinds of input they require, such as raw data or files.

Data hash algorithms

Data hash algorithms take raw bytes as input, without any additional metadata. They can be used by the data analyzer to assign the hash as an additional identifier of binary or text content, but it could also be used in other analyzers. For example, an XML analyzer might use canonicalization [55] before hashing to arrive at a hash that is possibly different from the one corresponding to the original binary data. Similarly, a bitmap analyzer might hash the raw pixel data stored in the image, allowing for the identification of images with the same pixels but stored in different formats.

File hash algorithms

These algorithms require an actual file or directory in a file system, and changing information such as the name of the file might change the resulting hash. One such example is the BitTorrent Info Hash, being in itself an SHA-1 hash of the BitTorrent `info` section describing a file or a directory and all its contents, usually stored as a part of a `.torrent` file.

These hash algorithms are used by the file analyzer. In the case of a directory, the semantics differ based on whether the hash is attached to a `nfo:FileDataObject` storing the directory, or the `nfo:Folder` inside. In the former case, the root directory should be represented in the same way as any other nested directory, while in the latter case, the directory is treated only as a collection of files. In the case of the BitTorrent Info Hash however, the name of the root directory may still be stored in the info section and, therefore, otherwise equivalent directories may result in a different hash when their names are different.

[29]Common encodings are hexadecimal, base32, or base64 [53], but they are only used by convention.

[30]These names are defined by the Named Information Hash Algorithm Registry located at `https://www.iana.org/assignments/named-information/named-information.xhtml`

[31]The proposed name `mh` could be used in `ni:` URIs as a fallback if the actual hash is not given a registered name. In that case, the hash is first wrapped in a multihash, storing the identifier of the hash algorithm, and then represented as a `ni:///mh;` URI.

[32]For common hashes, the URIs are taken from XML Signature Syntax and Processing [55], for example `http://www.w3.org/2000/09/xmldsig#sha1`.

4.4.5 Application

The core part of the application is an object outside the inspector, which accepts user-provided options, locates and opens the input files to be given to the inspector, configures it according to the provided options, and starts the analysis, storing the output at the provided output paths.

The application object itself is not a stand-alone program, rather it accepts an abstraction for controlling the external environment, such as writing messages to the user or opening and creating files. The options are given only as string arguments, as is the basic form of controlling applications from the command line[33]. The reason for this is to be able to execute it from various environments or processes, allowing the option to have a shared application interface for both the console and web environment, as is later discussed in section 5.1.

[33] The various supported command-line options are described in section 6.1.

5. Implementation

This chapter focuses on the actual implementation process, the choice of language and libraries, and other aspects which are pertinent when realizing the design outlined in the previous chapter. The complete source code of the software is accessible through GitHub[1].

5.1 Execution environment

The choice of environment, in which the software runs, determines what kind of operations the user will be able to perform with it, but restricting it to a specific kind of environment may also limit the range of possible files that the software can recognize.

As an example, restricting the software to a web-based application will most likely inhibit the application's ability to analyze large files, as they cannot be easily transferred over the internet, and may be subject to rate limits. It is possible to use a client-side web application to counter rate limits, but this again restricts the range of supported formats to those that can be analyzed via technologies supported by the browser. Desktop applications do not have this restriction, but there is still a decision to be made between a console-based application and a window-based application.

Based on these facts, it is better to build the application in a way that does not inherently restrict it to any of these environments but may be accessed from any of them via a common interface, as outlined in section 4.4.5. The only minimum requirements for the environment are text-based arguments and file manipulation, both of which can be provided or simulated via a console, window, or web-based environment. This makes it possible to develop a specific adaptation layer for any environment that meets these requirements.

Such a layer will be provided as a console application and as a client-side web application, both of which will use a common application API.

Providing input

Due to the environment-agnostic nature of the core application, communication between the user's file system and the application will also have to be done through the adaptation layer. This has the advantage of supporting additional methods of input if the environment provides them, such as directly via the standard input stream, or through the web. All these methods of input can be treated as files, providing a method to open the file in order to read the binary stream it stores. For this purpose, a file or a directory can be represented by an interface exposing common properties such as the name, date of creation, etc., and the list of contents in the case of a directory.

[1]`https://github.com/cermakmarek/SFI`

5.2 Language and framework

The choice of language is determined based on the intended platform, environment, and the purpose of the software, as some languages are better suited for specific tasks. In this case, C# was chosen, targeting .NET, for the following reasons:

- C# is a flexible and versatile language that supports both object-oriented programming and fast and safe raw memory access.
- C# is compiled into CIL, a portable intermediate language executable on many platforms. Ahead-of-time compilation to WebAssembly is also supported via the ASP.NET Blazor framework[2].
- .NET can run in all common execution environments, including console-based, window-based and web-based.
- External libraries and packages, e.g. for processing specific file formats, can be easily added via NuGet in Visual Studio.
- .NET supports P/Invoke, a technology allowing interoperability between managed and unmanaged code. This makes it possible to use system-specific functions to parse input files or to import libraries written in languages such as C or C++, if supported by the platform.

The software itself is separated into projects, the bulk of which target .NET Standard 2.0[3], a formal specification of .NET APIs that should be supported by concrete frameworks implementing it. .NET Standard 2.0 is implemented by all common frameworks based on .NET, including .NET Framework, .NET Core, Mono, Universal Windows Platform, Xamarin, and Unity, making it possible to incorporate parts of the software in projects targeting any of these platforms.

The remaining projects are concrete executable applications that target .NET 5[4], importing the base projects.

5.3 Choice of libraries

This section introduces the libraries that were used during implementation and reasons for using them.

dotNetRDF[5] dotNetRDF is a versatile and robust library for consuming, transforming, and producing RDF data, supporting a wide range of formats, as well as in-memory querying using SPARQL.

This library is internally used in all situations where RDF data is to be manipulated, but it is not exposed directly to analyzers. Instead, the linked node interface is used, making it possible to use a different library or implementation if needed.

[2]`https://learn.microsoft.com/cs-cz/aspnet/core/blazor/?view=aspnetcore-7.0`

[3]`https://learn.microsoft.com/cs-cz/dotnet/standard/net-standard?tabs=net-standard-1-0`

[4]`https://dotnet.microsoft.com/en-us/download/dotnet/5.0`

[5]`https://dotnetrdf.org/`

Ude.NetStandard[6] This is a .NET Standard port of the Mozilla Universal Charset Detector, containing heuristics for determining the encoding of text files. It is used as the primary encoding detector through a custom interface, making it possible to replace it when different detection methods are necessary.

SharpCompress[7] This library is used to decompress archives stored in ZIP, RAR, tar, gzip, or 7z formats, and exposes many properties obtainable from files within. The readers for individual archive formats all implement a single interface, making it possible to effortlessly add new archive formats without making additional analyzers.

TagLibSharp[8] This library supports the reading of arbitrary metadata from various file formats, expressed as "tags" – collections of pieces of metadata in various formats, such as ID3. It also supports browsing various codecs and media streams used by multimedia content.

This library is exposed as a single format and its analyzer, allowing retrieving a general representation of any file with metadata.

MetadataExtractor[9] This library is similar to TagLibSharp, but only for images. It represents an image as a list of "directories", where each directory consists of properties defined by specific metadata schemes, such as EXIF.

This library is used in the same way as TagLibSharp, with one format to retrieve the metadata "directories", which are then analyzed.

Vanara.PInvoke[10] This suite of libraries exposes the native Windows API using .NET classes and methods, internally utilizing the P/Invoke mechanism to call the proper API functions and marshal data.

It is used to parse several formats understood by Windows, such as the Portable Executable and Cabinet formats. Formats and analyzers using it are, however, unavailable on non-Windows platforms.

PeNet[11] This library supports loading executables using the Portable Executable format used by Windows. It is used by a single format, a managed alternative to the one using the Windows API.

NAudio[12] NAudio is a general-purpose audio library containing various abstractions for audio streams, with both managed and system implementations. It is used to recognize and load audio files, as well as read audio samples for additional analysis.

[6]`https://github.com/yinyue200/ude`

[7]`https://github.com/adamhathcock/sharpcompress`

[8]`https://github.com/mono/taglib-sharp`

[9]`https://github.com/drewnoakes/metadata-extractor-dotnet`

[10]`https://github.com/dahall/Vanara`

[11]`https://github.com/secana/PeNet`

[12]`https://github.com/naudio/NAudio`

DiscUtils[13] This is a library for loading various disk images, such as ISO, VHD, VDI etc., and supporting common file systems such as NTFS or FAT.

NPOI[14] This is a .NET version of Apache POI, supporting the various Microsoft Office formats, both OLE-based and XML-based.

OpenMcdf[15] This library is used by the OLE format to read files in the Microsoft Compound Document File Format.

SVG.NET[16] This library supports the loading of vector-based SVG files from XML, as well as rendering them as images.

PdfSharpCore[17] This is a .NET Standard library for parsing PDF documents.

Html Agility Pack[18] This library is capable of loading HTML documents.

BencodeNET[19] This library allows for encoding data in the Bencode format, used by `.torrent` files and required by the BitTorrent Info Hash.

Blake3.NET[20] This is a .NET implementation of the BLAKE3 cryptographic hash function.

SwfDotNet.IO[21] This library is used to open Shockwave Flash animations.

WarcProtocol[22] This library provides the parser for Web ARChive (WARC) files.

Aeon[23] This is an x86 DOS emulator, capable of executing DOS EXE and COM executables.

5.4 Classes and interfaces

This section describes common classes and interfaces representing abstractions introduced in section 4.4, as well as other abstractions which could be utilized by various formats and analyzers.

[13]`https://github.com/DiscUtils/DiscUtils`
[14]`https://github.com/dotnetcore/NPOI`
[15]`https://github.com/ironfede/openmcdf`
[16]`https://github.com/svg-net/SVG`
[17]`https://github.com/ststeiger/PdfSharpCore`
[18]`https://html-agility-pack.net/`
[19]`https://github.com/Krusen/BencodeNET`
[20]`https://github.com/xoofx/Blake3.NET`
[21]`https://www.nuget.org/packages/SwfDotNet.IO`
[22]`https://github.com/toimik/WarcProtocol`
[23]`https://github.com/gregdivis/Aeon`

5.4.1 Files and directories

Each item in any file system is represented by an instance of `IFileNodeInfo`, containing properties common to files and directories, and derived by interfaces `IFileInfo` and `IDirectoryInfo`, similarly to the relations between the base .NET classes `System.IO.FileSystemInfo`, `System.IO.FileInfo`, and `System.IO.DirectoryInfo`.

Anything that can be opened to provide an instance of `System.IO.Stream` is represented as an instance of `IStreamFactory`, a generalized version of `IFileInfo`.

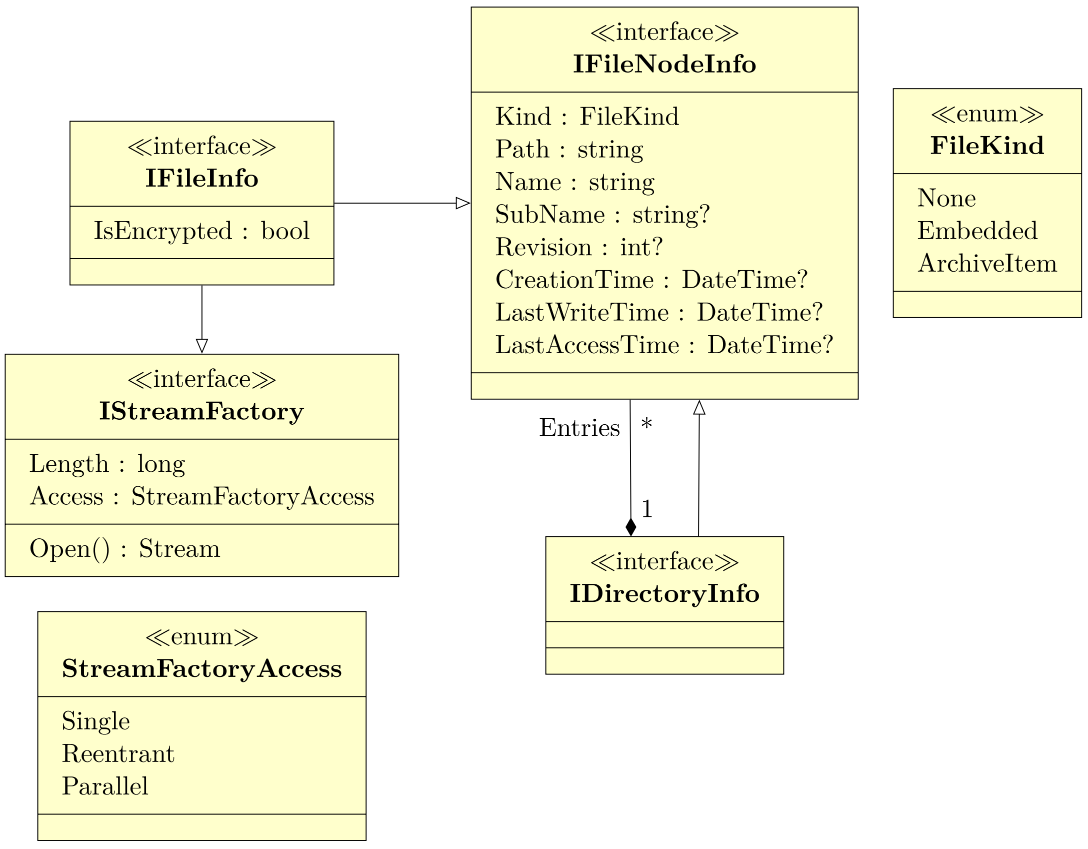


Figure 5.1: File and directory types hierarchy

5.4.2 Archives

Archives are represented in two ways: as an `IArchiveFile` when the implementation supports accessing the collection of entries as a whole, and as an `IArchiveReader` deriving from `IEnumerator<IArchiveEntry>` when the implementation can only move forward through the entries once. This is analogous to how archives are represented in SharpCompress.

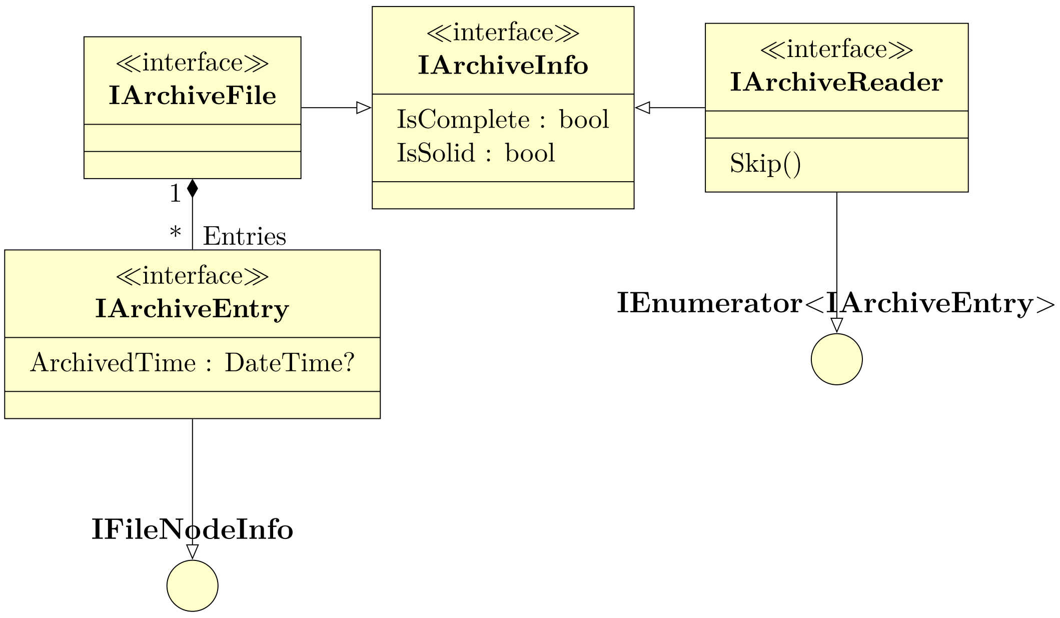


Figure 5.2: Archive types hierarchy

5.4.3 File formats

Each file format or a set of file formats, as designed in section 4.4.3 , is represented by an instance of the interface `IFileFormat`. Such an object supports retrieval of the media type of the concrete format and the usual extension via the methods `GetMediaType` and `GetExtension`. These methods require a value of a specific type supported by the object, to extract the information from. For example, a format representing various image formats supported by the `System.Drawing.Image` class requires an instance of `Image` to obtain the concrete format from, and it should also implement `IFileFormat<Image>`.

File formats with a concrete medium are represented by derived interfaces, such as `IBinaryFileFormat` for formats of byte sequences, and `IXmlDocumentFormat` for formats of XML documents, providing methods and properties useful for the relevant analyzers.

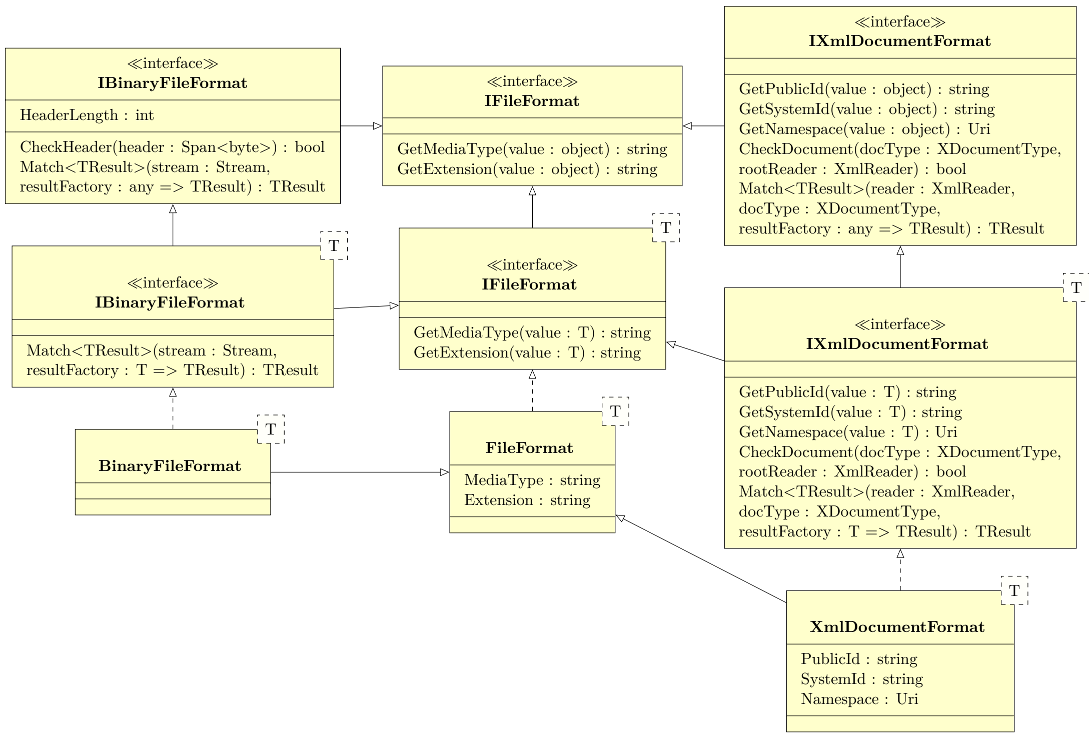

«interface»
IBinaryFileFormat
HeaderLength : int
CheckHeader(header : Span<byte>) : bool
Match<TResult>(stream : Stream, resultFactory : any => TResult) : TResult
«interface»
IFileFormat
GetMediaType(value : object) : string
GetExtension(value : object) : string
«interface»
IXmlDocumentFormat
GetPublicId(value : object) : string
GetSystemId(value : object) : string
GetNamespace(value : object) : Uri
CheckDocument(docType : XDocumentType, rootReader : XmlReader) : bool
Match<TResult>(reader : XmlReader, docType : XDocumentType, resultFactory : any => TResult) : TResult
T
«interface»
IBinaryFileFormat
Match<TResult>(stream : Stream, resultFactory : T => TResult) : TResult
T
«interface»
IFileFormat
GetMediaType(value : T) : string
GetExtension(value : T) : string
T
«interface»
IXmlDocumentFormat
GetPublicId(value : T) : string
GetSystemId(value : T) : string
GetNamespace(value : T) : Uri
CheckDocument(docType : XDocumentType, rootReader : XmlReader) : bool
Match<TResult>(reader : XmlReader, docType : XDocumentType, resultFactory : T => TResult) : TResult
T
BinaryFileFormat
T
FileFormat
MediaType : string
Extension : string
T
XmlDocumentFormat
PublicId : string
SystemId : string
Namespace : Uri

Figure 5.3: File format types hierarchy

5.4.4 RDF terms

To distinguish RDF terms suitable for different semantic purposes, value types `IndividualUri`, `PropertyUri`, `ClassUri`, `DatatypeUri`, and `GraphUri` are defined, all of them implementing the `ITermUri` interface. These types are used to denote items in a particular vocabulary, which is represented by `VocabularyUri`.

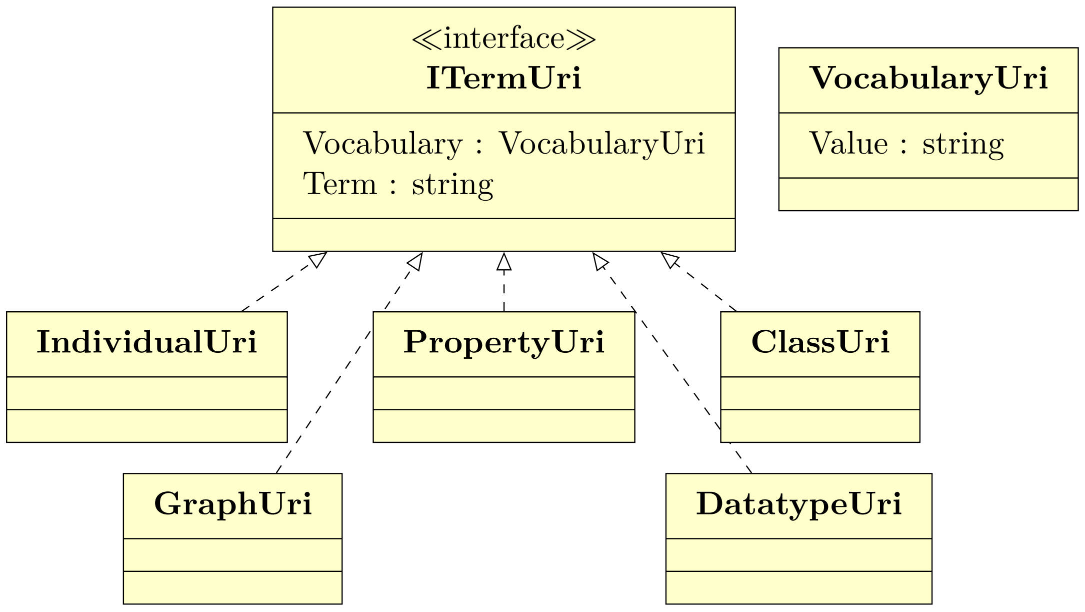


Figure 5.4: RDF term types hierarchy

How to statically define these terms is later described in section 5.5.1.

5.4.5 URI formatters

Complementary to the types for RDF terms, URI formatters are objects capable of creating a URI from an input value. These are used in similar situations as the types for RDF terms[24], and are similarly divided into `IIndividualUriFormatter`, `IPropertyUriFormatter`, `IClassUriFormatter`, `IDatatypeUriFormatter`, and `IGraphUriFormatter`.

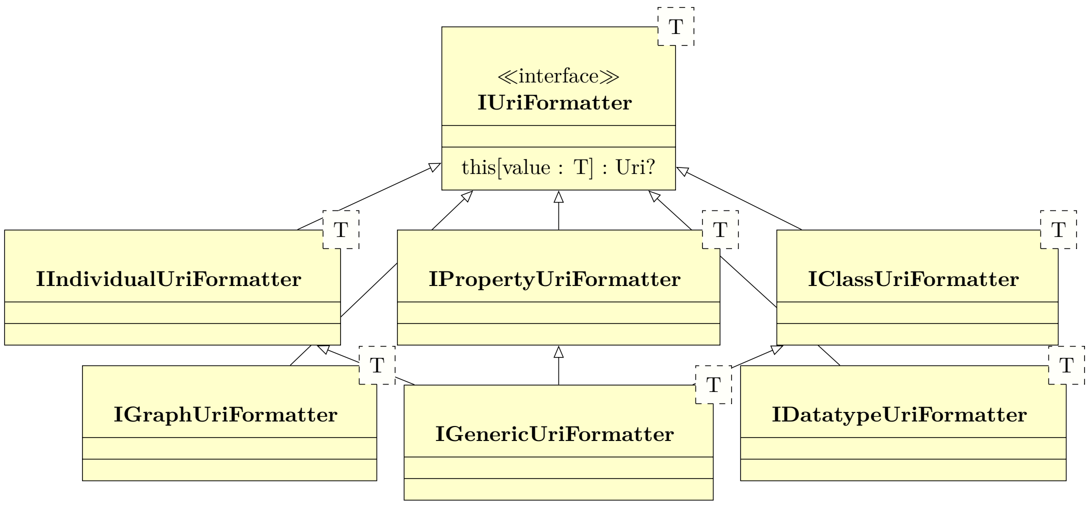


Figure 5.5: URI formatter types hierarchy

The `IGenericUriFormatter` interface is used in situations where the generated URIs may be usable as individuals, properties, or classes and it is not possible to further distinguish them.

5.4.6 Hash algorithms

Every hash algorithm, as described in detail in section 4.4.4, implements the interface `IHashAlgorithm`, storing information such as the name of the algorithm, but individual algorithms are further separated based on the type of input they accept – hash algorithms accepting arbitrary binary data implement `IDataHashAlgorithm`, hash algorithms using the properties of files or directories implement `IFileHashAlgorithm`, while hash algorithms for arbitrary objects are represented by `IObjectHashAlgorithm<T>`.

[24]These situations include cases where a particular part of a term's URI is fixed, while the rest depends on a value, for example UUID-based URIs with the fixed prefix `urn:uuid:` followed by a part based on the UUID. This may be represented as an `IUriFormatter<Guid>`, or using one of the more concrete interfaces.

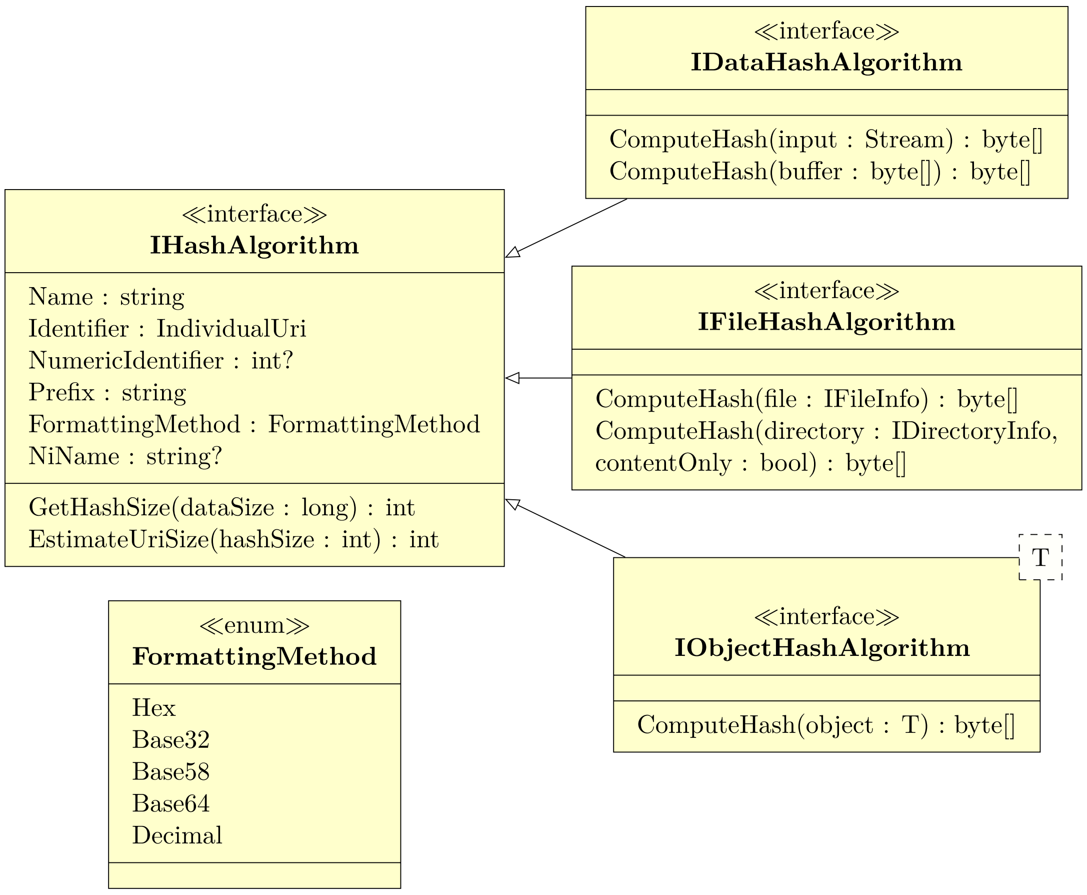

≪interface≫
IDataHashAlgorithm
ComputeHash(input : Stream) : byte[]
ComputeHash(buffer : byte[]) : byte[]
≪interface≫
IHashAlgorithm
Name : string
Identifier : IndividualUri
NumericIdentifier : int?
Prefix : string
FormattingMethod : FormattingMethod
NiName : string?
GetHashSize(dataSize : long) : int
EstimateUriSize(hashSize : int) : int
≪interface≫
IFileHashAlgorithm
ComputeHash(file : IFileInfo) : byte[]
ComputeHash(directory : IDirectoryInfo, contentOnly : bool) : byte[]
T
≪interface≫
IObjectHashAlgorithm
ComputeHash(object : T) : byte[]
≪enum≫
FormattingMethod
Hex
Base32
Base58
Base64
Decimal

Figure 5.6: Hash algorithm types hierarchy

5.4.7 Linked nodes

As introduced in section 4.4.1, a linked node is a write-only abstraction of an RDF resource. Linked nodes are manipulated as instances of `ILinkedNode`.

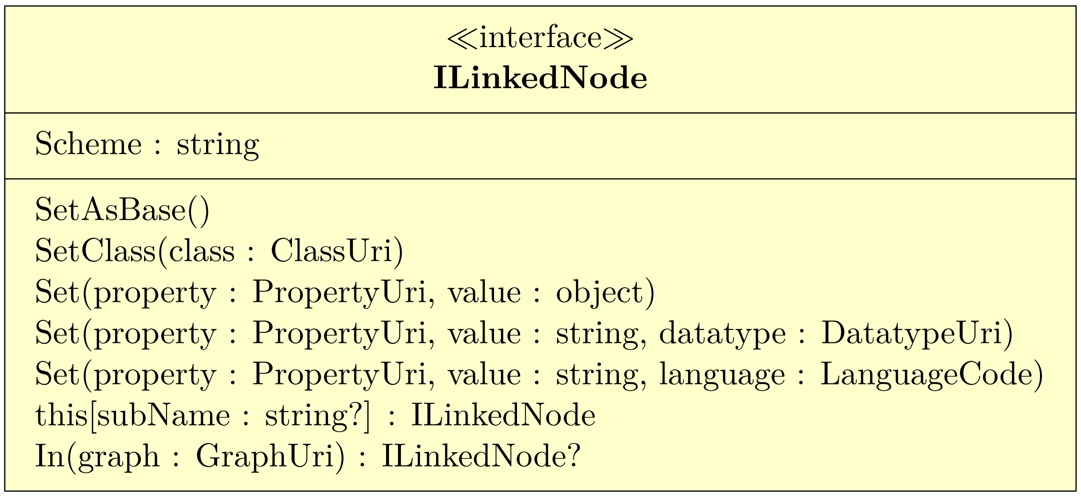

≪interface≫
ILinkedNode
Scheme : string
SetAsBase()
SetClass(class : ClassUri)
Set(property : PropertyUri, value : object)
Set(property : PropertyUri, value : string, datatype : DatatypeUri)
Set(property : PropertyUri, value : string, language : LanguageCode)
this[subName : string?] : ILinkedNode
In(graph : GraphUri) : ILinkedNode?

Figure 5.7: Linked node type

The `ILinkedNode` interface contains multiple overloads of the `Set` and `SetClass` methods with more complex parameters, allowing the usage of `IUriFormatter<T>` instead of fixed vocabulary items.

The `ILinkedNodeFactory` interface is used to create new instances of `ILinkedNode`.

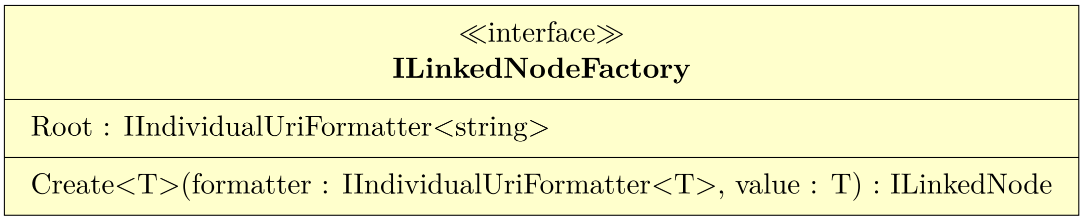

≪interface≫
ILinkedNodeFactory
Root : IIndividualUriFormatter<string>
Create<T>(formatter : IIndividualUriFormatter<T>, value : T) : ILinkedNode

Figure 5.8: Linked node factory type

The `Root` property can be used as a sort of default namespace for new entities formed from arbitrary strings as identifiers.

Both interfaces are meant to be implemented by concrete classes that encapsulate objects specific to the actual code manipulating RDF data, such as dotNetRDF.

5.4.8 Analyzers

Analyzers are objects capable of reading the properties of an entity and storing them through an instance of `ILinkedNode`, as introduced previously in section 4.4.2.

These objects are manipulated using two complementary interfaces, `IEntityAnalyzers` and `IEntityAnalyzer<in T>`[25]. The first interface is meant to be a general-purpose object capable of accepting any value, while the second interface is able to analyze only instances of `T`.

[25]This interface is contravariant on `T`, which means that every `IEntityAnalyzer<U>` is also a `IEntityAnalyzer<V>` if `V` is more specific than `U`.

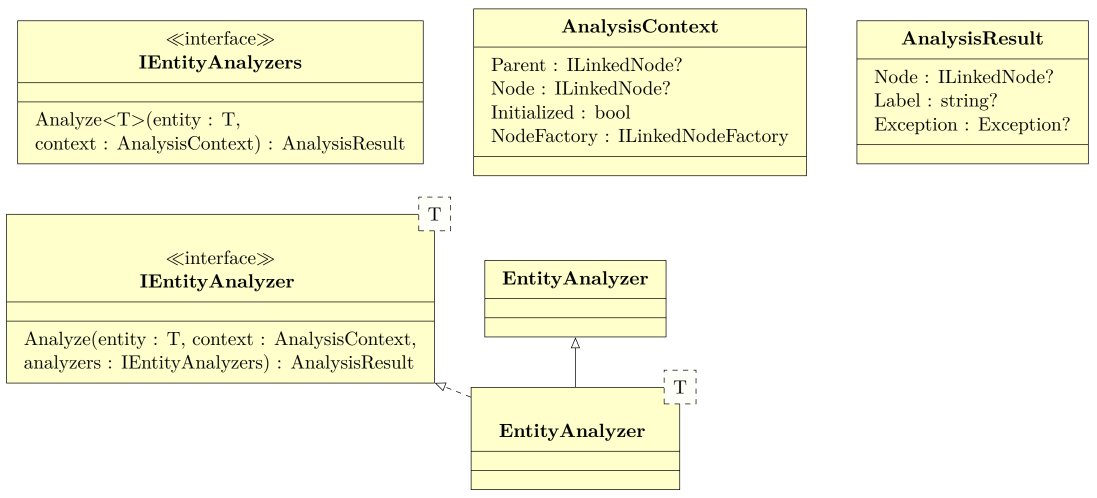


Figure 5.9: Entity analyzer types hierarchy

A potential implementation of `IEntityAnalyzers` could store a collection of objects, and when `Analyze` is called, use the first one that implements `IEntityAnalyzer<T>` for the type argument `T` specific to the `Analyze` call, passing itself through the `analyzers` parameter.

The result of the analysis is recorded in `AnalysisResult`, storing the `ILinkedNode` that should represent the analyzed entity, and other relevant information.

5.5 Challenges

In the following sections, some problems that arose during the implementation are introduced, as well as their solutions.

5.5.1 Using RDF vocabularies

One of the first challenges faced during development was how to "import" an RDF vocabulary to be used easily from code, in a manner comparable to how one would use it in an RDF language like Turtle.

The initial idea was to have an instance of a vocabulary, identified by a variable, and using indexing to obtain the representations of the terms in the vocabulary:

```
node.Set(Vocabularies.Rdf["value"], "value of the node");
```

Listing 5.1: Initial idea of assigning a property of a node

This is an approach similar to how derived linked nodes are created, but it has downsides: indexing the vocabulary requires creating the URI each time, or using caching, and is prone to mistakes when the name has to be typed each time. Additionally, it is not easy to replace a term with a different one by modifying the vocabulary, since the URI is already given by the string index.

It is better to represent each term by an actual field, having the additional advantage that removing a term leads to a compile-time error. A custom attribute is used to mark from which vocabulary the term stored in the field should come.

```
// The vocabularies are not normally needed; only their URIs
   ↪are kept as constants.
public static class Vocabularies
{
    public static class Uri
    {
        public const string Rdf = "http://www.w3.org/1999/02/22-
           ↪rdf-syntax-ns#";
        public const string Rdfs = "http://www.w3.org/2000/01/
           ↪rdf-schema#";
        public const string Owl = "http://www.w3.org/2002/07/owl
           ↪#";
    }
}

// All imported RDF properties are kept in a single class, to
   ↪easily keep track of which properties are used. This is,
   ↪however, not necessary.
public static class Properties
{
    // A custom attribute is used to link to the vocabulary.
    [Uri(Vocabularies.Uri.Rdf)]
    public static readonly PropertyUri Type;
    [Uri(Vocabularies.Uri.Rdf)]
    public static readonly PropertyUri Value;

    [Uri(Vocabularies.Uri.Rdfs)]
    public static readonly PropertyUri Label;

    // The local name of the term can be specified as well.
    [Uri(Vocabularies.Uri.Owl, "sameAs")]
    public static readonly PropertyUri IsSameAs;

    static Properties()
    {
        // This method enumerates all static properties and
           ↪assigns their values based on the Uri attribute,
           ↪using reflection.
        typeof(Properties).InitializeUris();
    }
}
```

Listing 5.2: Example of a static type with predefined RDF properties

5.5.2 Type-introducing return values

All analyzers implement the `IEntityAnalyzer<in T>` interface where `T` denotes the type of objects the analyzer can handle, as defined in section 5.4.8. This makes testing whether a particular analyzer can handle a specific type very easy, but there are situations where the concrete type is not easily retrievable:

```
public interface IFormatObject
{
    string Extension { get; }
    string MediaType { get; }
    IFileFormat Format { get; }
}

public interface IFormatObject<out T> : IFormatObject
{
    T Value { get; }
}
```

Listing 5.3: The `IFormatObject` interfaces

While it is possible to obtain the value as an object, by casting to `IFormatObject<object>`, the intended type `T` is still not retrievable, without using reflection. The reason for having to obtain the proper type at all is that some classes may implement two interfaces which are handled by two separate analyzers, and it may be necessary to pick the intended interface to analyze.

The adopted solution was to use a "generic callback" to retrieve the result, as an instance of `IResultFactory`:

```
public interface IResultFactory<out TResult, in TArgs>
{
    ITask<TResult> Invoke<T>(T value, TArgs args);
}

public interface IFormatObject
{
    string Extension { get; }
    string MediaType { get; }
    IFileFormat Format { get; }

    ValueTask<TResult> GetValue<TResult, TArgs>(IResultFactory<
       ↪TResult, TArgs> resultFactory, TArgs args);
}
```

Listing 5.4: The `IResultFactory` interface and its use in `IFormatObject`

Inside `GetValue`, `resultFactory.Invoke` should be called by the implementation, which has access to `T` and can provide it to the method, alongside any arguments the caller specified. The calling code is free to provide any implementation of `IResultFactory` to handle the value.

5.5.3 Caching temporary objects

File-based hashes, such as the BitTorrent Info Hash, usually need to hash the data inside the file as a part of creating their own hash, but at this point the data might already be inaccessible. Some archive formats only support sequential reading through the files, meaning parts of the data relevant to the file hash have to be computed alongside the other data hashes, cached, and later retrieved by the file hash.

This poses a problem how to store the hash, or specifically how to pick the cache by which the hash is cached. It is generally not possible to use the file object itself, because there is no guarantee that the same object instance will be used to represent the same file the next time it is retrieved.

This is solved using the `IPersistentKey` interface, storing two keys that persist beyond the intended lifetime of the object:

```
public interface IPersistentKey
{
    object ReferenceKey { get; }
    object DataKey { get; }
}
```

Listing 5.5: The `IPersistentKey` interface

Such an object provides two keys, `ReferenceKey` which is compared for identity, i.e. using `Object.ReferenceEquals`, and `DataKey` compared for equality using `Object.Equals`. The first object should be a complex object which should persist in memory for as long as possible, for example the object representing the archive a file is in. The second object should be a simpler value that uniquely identifies the original object inside within a single `ReferenceKey`, for example a string storing the path of the file in the archive.

5.5.4 Adapting for the browser

Via Blazor, it is possible to write code in .NET that can be executed on both the desktop and in the browser. There are, however, significant differences between the two environments, which require some adaptations to parts of the code:

- All waiting must be performed purely by asynchronous methods. It is not possible to use methods like `Task.Wait` to pause the current thread. As a consequence, every I/O operation must be non-blocking and must use `await`, requiring almost every method to return a task, either `ValueTask` for optimization, `ITask`[26] when covariance is needed, or a `Task` otherwise.
- Multithreading may not be supported, in which case all parallel code including `Task.Run` will be executed sequentially. The code must be adapted to work with this restriction.
- Some hashes, Windows-specific formats, and native libraries are not supported, having to be disabled.

5.5.5 Circumventing automatic conversions of URIs

In some cases, .NET performs automatic conversions of URIs in regards to percent-encoded characters:

[26] Imported from the package MorseCode.ITask.

```
var uri1 = new Uri("urn:á");
Console.WriteLine(uri1.AbsoluteUri); // urn:%C3%A1
Console.WriteLine(uri1.ToString()); // urn:á

var uri2 = new Uri("urn:%C3%A1");
Console.WriteLine(uri2.AbsoluteUri); // urn:%C3%A1
Console.WriteLine(uri2.ToString()); // urn:á
```

Listing 5.6: Behaviour of `Uri` in .NET

In some situations, dotNetRDF calls `Uri.ToString()` to obtain the string value of the URI, so if the automatic conversion to IRI is not required, it can be overridden:

```
public class EncodedUri : Uri
{
    public EncodedUri(string uriString) : base(uriString)
    {

    }

    public EncodedUri(string uriString, UriKind uriKind) : base
        ↪(uriString, uriKind)
    {

    }

    public override string ToString()
    {
        return IsAbsoluteUri ? AbsoluteUri : OriginalString;
    }
}
```

Listing 5.7: The `EncodedUri` class

Similarly, `Uri.OriginalString` may be returned if its validity can be ensured.

5.5.6 Opening Cabinet archives

The Cabinet format used by older Windows installations can be opened using Windows API functions, also exposed in `Vanara.PInvoke.Cabinet`. The contents of the archive are iterated using a push-based approach; the `FDICopy` function repeatedly calls a specified callback function in different events, such as when an archive is opened, a file is encountered, or the archive ends. The archive analyzer, however, enumerates the archive contents in a pull-based way, through `IEnumerable`.

The solution is to use a thread to call `FDICopy`, and communicate with it using a channel – from within the thread, files are pushed to the channel, while outside the thread, they are pulled from the channel. The thread is paused until the channel is emptied.

The callback in `FDICopy` is expected to return a writable stream when a file in the archive is requested to be, while the data analyzer expects to read from the stream. The same approach is used to solve this issue – a channel is used to store the data requested to be written, and the thread remains paused until the channel is emptied.

During the implementation, several bugs were also found in Vanara.PInvoke regarding interoperability with the native Cabinet API, which were reported and fixed by the maintainers of the library.

5.5.7 Forking the XML reader

In .NET, XML data is at its core consumed using the `XmlReader` abstract class. This is the class used by the XML analyzer, `IEntityAnalyzer<XmlReader>`, but the class only supports moving forward in the XML stream, not backward. This poses an issue when there are multiple formats recognizable from the XML file, for example when an SVG file is also annotated in some general way, for example by attributes on the root element.

In this case, multiple formats have to read data coming from the same XML reader, but the data must be read only once. This is achieved by capturing "snapshots" of the current state of the `XmlReader`, including information about the current node or its attributes, which are sent via channels to each of the formats during reading. The channel assigned to each format is viewed as a `XmlReader`, moving through the channel and invoking properties from the latest state.

6. Documentation

In the following sections, documentation for the software is provided. In section 6.1, information on how to operate the software as a user in general is provided, for both the console and web application. In section 6.2, the overall structure of the solution is described, including the list of projects, namespaces, and important classes, and section 6.3 specifies how can plugins be added to the application. Lastly, in section 6.4, instructions on how to publish and run the included web application are provided.

6.1 User documentation

The software may be downloaded from the GitHub repository[1] as the console application, or tried out in the browser[2]. The console version is built for the `win-x64` platform, and it is also possible to build the application from source code, requiring at least .NET 5.0[3] and C# 9.

Both the console and web application are controlled through a shared command-line interface. The usage of the application and the supported options can be displayed by passing `-?` as the command-line option.

```
Usage: (describe|search|list) [options] input... output
```

Listing 6.1: Parameters of the application

The application operates in three modes: `describe`, `search`, and `list`. In `list`, it only displays a list of supported components and their properties, which can be used to configure them further. In `describe`, the application loads a collection of input files and describes them using RDF, saving the output in a desired RDF serialization format to the last file specified. In `search`, the application requires a list of SPARQL queries in addition to the input files and evaluates them on the RDF descriptions, saving the output in a desired SPARQL results serialization format to the last file specified.

Input files support wildcard characters like `?` and `*` to select multiple files at once[4]. In the console-based application, if `-` is used as one of the input files, the standard input of the process is used as the input and is analyzed as if it were a file. Additionally, in both applications, `-` may also be used as the output file, in which case the RDF data is written to the standard output in the case of the console application, or to the log in the web application. It is also possible to specify `NUL` (case-insensitive) or `/dev/null` as the output, which discards any RDF data.

Additional arguments given to the application must be passed as options before the input files, beginning with `-` for the short form or `--` for the long form:

[1] `https://github.com/cermakmarek/SFI/releases`

[2] `https://cermakmarek.github.io/sfi/app/`

[3] `https://dotnet.microsoft.com/en-us/download/dotnet/5.0`

[4] The `?` character stands for a single arbitrary character, while `*` stands for any number of arbitrary characters.

Short form	Long form	Argument	Description
`q`	`quiet`		No logging messages, normally sent to the standard error, are produced.
`i`	`include`	pattern	From the previously excluded components, includes those matching the pattern.
`e`	`exclude`	pattern	From the previously included components, excludes those matching the pattern.
`f`	`format`	extension or MIME type	Sets the RDF serialization format of the output (ttl, jsonld, rdf etc.) in case of `describe`, or the SPARQL results format in case of `search`. Also deduced from the output file extension.
`h`	`hash`	pattern	Set the primary data hash algorithm. The algorithm is also included as a component. Only a sufficiently collision-resistant hash algorithm should be used.
`c`	`compress`		Enables gzip compression for the output.
`m`	`metadata`		Adds annotation metadata to the output.
`d`	`data-only`		Only treats the input files as plain data, without file information.
`u`	`ugly`		Use compact, non-pretty mode of RDF output writing.
`b`	`buffered`		Buffer the RDF triples in memory before writing them to the output all at once.
`r`	`root`	URI	Sets the URI prefix that is used for unique entities which do not have a stable identifier. Without this option, only blank nodes are used. A prefix like `urn:uuid:` or a Skolem IRI prefix, under `/.well-known/genid/`, is recommended.
`s`	`sparql-query`	file	The given file is executed as a SPARQL query, in case of `describe` for selecting files, or in case of `search` to query for information from the description.

Table 6.1: All command-line options of the application

Examples

`describe dir/* out.ttl`
: Describes all files in `dir` using the default components, and saves the RDF output to `out.ttl`.

`describe -d -h sha1 dir out.ttl`
: The same as above, but only loads the files in the directory as data (`-d`), without storing their names or other metadata. In addition to that, the SHA-1 hash algorithm is used to produce `ni:` URIs for content.

`describe -f rdf dir -`
: Same as the first example, but writes the RDF description as RDF/XML to the standard output.

`describe -b -f jsonld dir -`
: Writes the RDF description in JSON-LD instead. This requires buffering the output (`-b`).

`describe -r urn:uuid:  dir -`
: Does not use blank nodes to identify entities, instead using URIs starting with `urn:uuid:`.

`describe -x *-hash:* -i data-hash:sha1 dir -`
: Does not use any of the supported hash algorithms, with the exception of SHA-1, to describe data.

More detailed examples of the usage of the application modes, as well as the expected output, can be found in section 7.5.

6.1.1 Configuring components

Various configurable objects, understood by the application, such as analyzers, file formats, or hash algorithms, are collectively stored in *component collections* and can be included or excluded from them via the `-i` and `-x` options.

When present in a collection, a component receives its *compound identifier*, formed from the name of the collection, and the base name of the component, joined using `:`. For example, components in the analyzer collection are identified using the `analyzer:` prefix, and can be collectively removed using `-x analyzer:*`[5].

It is possible a single component may be included in multiple collections at once, for example data hash algorithms may be used by the data analyzer for hashing arbitrary bytes, using the `data-hash:` prefix, or by the image analyzer for analyzing raw pixel data, using the `pixel-hash:` prefix. Using the proper pattern, a particular hash algorithm may be configured to be present in only one collection, both or none of them.

[5]The wildcard character `*` represents any number of characters, while `?` represents a single arbitrary character.

Analyzers form their base name from the supported type of analyzed objects, for example the analyzer of images has the identifier `analyzer:image`. If the type is located in a foreign assembly, the name of the assembly is also used as a part of the base name, for example the analyzer of wave streams from the core NAudio library has the identifier `analyzer:n-audio.core.wave-stream`.

Formats use the MIME type of the recognized files as their base names, for example the identifier of the XML format is `data-format:application/xml`. If the MIME type of the format is not known, or if the format does not represent a single concrete format, the special unregistered MIME type prefix `application/x.obj.` is used, followed by the base name of the analyzer that would recognize this object, for example in `data-format:application/x.obj.image`.

The base names of other components do not follow any particular pattern.

Some components may have configurable properties that can be browsed by running the application in the `list` mode. The identifier of a component's property is formed by joining the compound identifier of the component and the name of the property by `:`, and can be used as an option by prefixing it with `--`. For example, the estimate of the base size of a single triple in bytes in a triple store[6] can be configured using `--analyzer:stream-factory:triple-size-estimate`.

Additional components of arbitrary types may be added by the user in the form of plugins; see section 6.3.1 for instructions on how to load plugins into the application.

Available components

This is the list of available components and their properties:

`analyzer:file-node-info`
This is the file analyzer, as designed in section 4.4.2, accepting arbitrary files and directories. This component holds the collection of file hash algorithms, prefixed `file-hash:`.

`analyzer:stream-factory`
This is the data analyzer, as designed in section 4.4.2, accepting any source of data or sequence of bytes. This component holds the collection of binary formats, prefixed `data-format:`, and data hash algorithms, prefixed `data-hash:`.

`analyzer:stream-factory:file-size-to-write-to-disk`
The minimum size at which the data is written to a temporary file on disk instead of being stored in memory.

`analyzer:stream-factory:min-data-length-to-store`
The minimum size at which to consider storing data directly in the URI, instead of using one of its hashes to identify it.

[6]This property is used by the data analyzer to decide whether to use the raw data or the primary hash as the means of identifying an entity, based on whichever takes less space.

`analyzer:stream-factory:triple-size-estimate`
: An estimate of the size of a triple in a data store, used to evaluate whether describing the data using hashes would be more efficient than storing it in the URI.

`analyzer:stream-factory:max-depth-for-formats`
: The maximum depth the data is allowed to be as an entity in a hierarchy in order to attempt to analyze formats.

`analyzer:data-object`
: This is the data object analyzer, as designed in section 4.4.2. Excluding this component will disable describing data, but individual media objects will still be analyzed.

`analyzer:data-object:label-size-suffix-digits`
: The number of decimal digits used to format the data size in the label.

`analyzer:format-object`
: This is the format object analyzer, as designed in section 4.4.2. Excluding this component will prevent any format analysis.

`analyzer:format-object:label-size-suffix-digits`
: The number of decimal digits used to format the data size in the label.

`analyzer:xml-reader`
: The analyzer of XML documents. This component holds the collection of XML formats, prefixed `xml-format:`.

`analyzer:x509-certificate`
: The analyzer of X.509 certificates.

`analyzer:x509-certificate:describe-extensions`
: Whether to use the certificate's extensions to provide an additional description.

`analyzer:image`
: The analyzer of arbitrary images. This component holds the collection of image hash algorithms, prefixed `image-format:`, and pixel data hash algorithms, prefixed `pixel-hash:`[7].

`analyzer:image:make-thumbnail`
: Whether to produce a small thumbnail data: node from the image.

`analyzer:read-only-list.metadata-extractor.directory`
: The analyzer of metadata directories, produced by MetadataExtractor from image files.

`analyzer:metadata-extractor.exif-directory-base`
: The analyzer of EXIF metadata in MetadataExtractor directories.

[7]This collection contains the duplicates of all hash algorithms under `data-hash:`, since they are the same algorithms, just used for hashing the actual pixel data

`analyzer:metadata-extractor.xmp-directory`
: The analyzer of XMP metadata in MetadataExtractor directories.

`analyzer:tag-lib-sharp.file`
: The analyzer of TagLibSharp files, as containers of tags.

`analyzer:tag-lib-sharp.xmp-tag`
: The analyzer of XMP TagLibSharp tags.

`analyzer:svg.svg-document`
: The analyzer of SVG documents from SVG.NET.

`analyzer:n-audio.core.wave-stream`
: The analyzer of audio streams from NAudio.

`analyzer:n-audio.core.wave-stream:create-spectrum`
: Whether to produce spectrogram images from the audio.

`analyzer:swf-dot-net.io.swf`
: The analyer of Shockwave Flash animations from SwfDotNet.IO.

`analyzer:npoi.poi-document`
: The analyzer of OLE-based documents from NPOI.

`analyzer:npoi.ooxml.poixml-document`
: The analyzer of OOXML-based documents from NPOI.

`analyzer:pdf-sharp-core.pdf-document`
: The analyzer of PDF documents from PdfSharpCore.

`analyzer:html-agility-pack.html-document`
: The analyzer of HTML documents from the Html Agility Pack.

`analyzer:archive-file`
: The analyzer of whole archives from SharpCompress.

`analyzer:archive-reader`
: The analyzer of archives from SharpCompress, read sequentially.

`analyzer:cabinet-archive`
: The analyzer of Cabinet archives. The analyzed archive is wrapped and handled to `analyzer:archive-reader`.

`analyzer:disc-utils.core.file-system`
: The analyzer of file systems from DiscUtils.

`analyzer:module`
: The analyzer of Windows executable or resource modules.

`analyzer:win-version-info`
: The analyzer of Windows version resources, in the `VS_VERSIONINFO` format[8].

[8]`https://learn.microsoft.com/en-us/windows/win32/menurc/vs-versioninfo`

`analyzer:dos-module`
The analyzer of DOS executables, using Aeon to execute them.

`analyzer:dos-module:emulate`
Whether to attempt to run the executable to obtain additional data.

`analyzer:dos-module:console-encoding-name`
The encoding used to decode texts produced by the module.

`analyzer:dos-module:instruction-step`
The number of instructions to emulate in each step.

`analyzer:dos-module:instruction-limit`
The limit on the total number of instructions after which emulation is stopped.

`analyzer:delphi-object`
The analyzer of Delphi DFM objects, stored as resources in executables.

`analyzer:open-mcdf.compound-file`
The analyzer of OLE compound files from OpenMcdf.

`analyzer:package-description`
The analyzer of packages using the `FILE_ID.DIZ` file for description.

`analyzer:rdf-xml-analyzer.document`
The analyzer of RDF/XML documents.

`analyzer:async-enumerable.toimik.warc-protocol.record`
The analyzer of WARC files from WarcProtocol.

`analyzer:vanara.p-invoke.shell32.url.uniform-resource-locator`
The analyzer of internet links from Vanara.PInvoke.

`analyzer:vanara.p-invoke.shell32.shell32.shell-link-w`
The analyzer of shell links from Vanara.PInvoke.

`data-format:application/x-delphi-form:default-encoding-name`
The encoding used to read strings.

`data-format:text/html:default-encoding-name`
The encoding chosen for HTML by default.

`data-hash:md5`
The MD5 hash algorithm.

`data-hash:sha1`
The SHA-1 hash algorithm.

`data-hash:sha256`
The SHA-256 hash algorithm.

`data-hash:sha384`
The SHA-384 hash algorithm.

`data-hash:sha512`
: The SHA-512 hash algorithm.

`data-hash:bsha1256`
: A variable-length hash algorithm producing SHA-1 hashes from 256 KiB-long chunks of the hashed data.

`data-hash:blake3`
: The BLAKE3 hash, implemented by Blake3.NET.

`data-hash:crc32`
: The CRC32 checksum algorithm.

`data-hash:crc64`
: The CRC64 checksum algorithm.

`data-hash:xxh32`
: The xxHash-32 checksum algorithm.

`data-hash:xxh64`
: The xxHash-64 checksum algorithm.

`image-hash:dhash`
: A dHash[9] low-frequency image hash algorithm.

`file-hash:btih`
: The BitTorrent Info Hash, using BencodeNET.

`file-hash:btih:block-size`
: The size of the individual file chunks in bytes, 256 KiB by default.

Furthermore, there are also components for generally recognized formats accepted by the analyzers[10], and the specific formats that are supported by the application also have a corresponding `*-format:` component[11].

[9]This hash works by encoding the difference between adjacent pixels in a downscaled version of the image. There is not a single universal dHash variant – the one used by this application works on two downscaled images, 9x8 and 8x9, the first one used for horizontal differences and the second for vertical differences, both based on the brightness of the pixels, in the HSB color model. The difference maps are alternated and traced using a third order Hilbert curve. Additionally, the bit chosen for equal brightness in the horizontal difference map is 0, while the vertical map uses 1, forming an alternating bit pattern for same-color image sections.

[10]Such as `data-format:application/x.obj.image`, `data-format:application/x.obj.n-audio.core.wave-stream`, `data-format:application/x.obj.read-only-list.metadata-extractor.directory`, `data-format:application/x.obj.tag-lib-sharp.file`, or `data-format:application/x.obj.x509-certificate2`.

[11]These are `image/svg+xml`, `audio/vnd.wave`, `application/ogg`, `application/vnd.adobe.flash-movie`, `application/msword`, `application/vnd.ms-excel`, `application/vnd.openxmlformats-package`, `application/vnd.openxmlformats-officedocument.spreadsheetmlsheet`, `application/vnd.openxmlformats-officedocument.wordprocessinml.document`, `application/pdf`, `text/html`, `application/zip`, `application/gzip`, `application/vnd.rar`, `application/vnd.ms-cab-compressed`, `application/x-ms-compress-sz`, `application/x-7z-compressed`, `application/x-tar`, `application/x-iso9660-image`, `application/x-udf`, `application/x-dosexec`, `application/x-msdos-program`, `application/x-msdownload`, `application/x-msdownload;`

Examples

`-x *-format:* -i *-format:image/*`
: Excludes all file formats from the list of components, but keeps specific image formats.

`-x * -i analyzer:stream-factory -i analyzer:data-object`
: Only allows for the analysis of actual data, not files.

`--analyzer:stream-factory:max-depth-for-formats ""`
: Sets this property value to null, disabling depth checks.

6.1.2 Using SPARQL

Using the `--sparql-query` option, it is possible to execute custom SPARQL queries during the process, and match various entities handled by the analyzers.

Each query is executed at various points during the analysis. The data available to the query differs based on the presence of the `--buffered` option: if the option is present, the query operates on the whole graph, while if the option is not present, only a small section of the data, usually enough to describe a single entity, is used.

Search

In the `search` mode, the `--sparql-query` option should point to a `SELECT` or `ASK` query. When a query is evaluated, its results are added to an internal storage, which is serialized to the output file when the process stops.

The evaluation of a query in this mode may also stop the process prematurely if one of these conditions succeeds:

- The query uses `ASK`, and its result is determined to be `true`.
- The query uses `LIMIT`, and the number of results exceeds the limit. The process will be stopped in this case only if there are no other queries that may yet produce results, such as queries without `LIMIT`.

```
PREFIX schema: <http://schema.org/>

ASK WHERE {
  [] schema:encodingFormat <https://w3id.org/uri4uri/mime/image
    ↪/png> .
}
```

Listing 6.2: An example `ASK` query determining the presence of a PNG image

`format=le`, `application/x-msdownload;format=ne`, `application/vnd.microsoft.portable-executable`, `application/x-delphi-form`, `application/x-ole-storage`, `application/xml`, `application/json`, `application/rdf+xml`, `application/warc`, `text/x-uri`, and `application/x-ms-shortcut`.

```
PREFIX nfo: <http://www.semanticdesktop.org/ontologies
    ↪/2007/03/22/nfo#>
PREFIX nie: <http://www.semanticdesktop.org/ontologies
    ↪/2007/01/19/nie#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX schema: <http://schema.org/>

SELECT DISTINCT ?name ?w ?h
WHERE {
  [
    nfo:fileName ?name ;
    nie:interpretedAs/dcterms:hasFormat [
      a schema:ImageObject ;
      nfo:width ?w ;
      nfo:height ?h
    ]
  ] .
}
```

Listing 6.3: An example `SEARCH` query that retrieves the names of image files and their dimensions

In addition to the usual formats for storing SPARQL results, it is also possible to save them as a new SPARQL query[12] that uses `VALUES` to replicate the same results.[13]

File extraction

In the `describe` mode, the `--sparql-query` option should point to a `SELECT` query, which will be used to mark entities that should be matched and extracted if they are backed by binary data. The query should have a variable `?node`, which is compared against the node representing the currently analyzed entity, extracting it as a file if the nodes are equal.

The name of the file can be determined by assigning the `?path_format` variable in the query, which has the default value `"${name}${extension}"`. Other properties related to the file may be substituted in `?path_format`, including `${media_type}` or `${size}`.

```
SELECT ?node ?path_format
WHERE {
  ?node ?p ?o .
  BIND("extracted/${name}${extension}" AS ?path_format)
```

[12]Using the `application/sparql-query` media type, or the `sparql` or `rq` extension.

[13]This may be useful to prepare a query for file extraction in a two-pass style, by first using `-b` to enable buffered mode in the `search` mode, while providing a SPARQL query made for extraction. This evaluates it on the whole graph, having a complete overview of the RDF data as opposed to running it in the non-buffered mode. Using this pre-computes the nodes produced by the extraction query in the `describe` mode, which might otherwise finish processing the extracted entities before they can be matched by the query, at which point they are no longer extractable.

```
}
```

Listing 6.4: An example query that matches all entities; extracting them to the "extracted" folder according to their name and extension

```
PREFIX nfo: <http://www.semanticdesktop.org/ontologies
   ↪/2007/03/22/nfo#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX schema: <http://schema.org/>

SELECT DISTINCT ?node
WHERE {
  ?node dcterms:hasFormat [
    a schema:ImageObject ;
    nfo:width 256 ;
    nfo:height 256
  ]
}
```

Listing 6.5: An example query that matches all 256x256 images

6.2 Programmer documentation

This section describes the structure of the code and the various projects it consists of. The whole solution is divided into several projects, based on their primary purpose and dependencies. This allows to distribute the projects as packages to be used as libraries in other solutions, selected based on the individual needs of those solutions.

The documentation of all classes and methods is available in structured form, generated from the XML documentation comments included in the code[14], while the individual projects are described in the following sections.

6.2.1 `SFI`

The `SFI` project is the core part of the software, defining the common interfaces used in the other projects, as well as core analyzers and formats whose implementation does not depend on any external libraries outside of .NET. It defines types from these namespaces:

`SFI` This namespace contains various utility classes such as `DataTools`, `TextTools`, and `UriTools`, as well as classes containing extension methods intended to be used from all other components.

`SFI.Analyzers` This namespace is intended to store all analyzers, in this project or others. In the core project, the only defined analyzers are for objects whose types are defined in .NET or in the core project, such as `FileAnalyzer`, `DataAnylzer`, `XmlAnalyzer`, or `X509CertificateAnalyzer`.

[14] `https://cermakmarek.github.io/sfi/docs/`

`SFI.Formats` This namespace is similar to the previous one but stores classes used for defining and parsing formats. Aside from common interfaces and related components defined here, it defines only formats that can be used from pure .NET, such as `XmlFileFormat`, `ZipFileFormat`, and `X509CertificateFormat`.

`SFI.Services` This namespace contains specialized interfaces to be used for communication between formats and analyzers or other components in the solution, such as `IFileNodeInfo`, `IFormatObject`, `IPersistentKey`, `ILinkedNode`, `ILinkedNodeFactory`, `IEncodingDetector`, and similar, as well as their base implementations.

`SFI.Tags` Tags, in this context, are usually small objects intended to be applied to existing objects via other means, providing extended description of them beyond what their original classes support.

In this project, the tags that are defined are `IImageTag` and `IImageResourceTag`, usually applied to images via the `Image.Tag` property, providing information about the origin of the image or the allowed operations.

`SFI.Tools` This namespace hosts various specialized utility classes for general use and for I/O and XML operations. It also exposes the collection of built-in hash algorithms using the `BuiltInHash` class, the `PersistenceStore` class, used for storing values attached to instances of `IPersistentKey`, and the `EncodedUri`, which should be used instead of the base `Uri` class in all situations to control its formatting, as justified in section 5.5.5.

`SFI.Vocabulary` This namespace provides datatypes used for defining RDF vocabularies, such as `ClassUri`, `PropertyUri` or `LanguageCode`, as well as storing the common vocabulary terms in the static classes `Individuals`, `Classes`, `Properties`, and `Datatypes`, as described in section 5.5.1, to be used from code easily.

6.2.2 `SFI.Accessories`

This project provides a concrete implementation of the core classes and interfaces. As such, it imports several external projects, like dotNetRDF for the RDF output in `LinkedNodeHandler`, or UDE for encoding detection in `UdeEncodingDetector`, while the core project can theoretically work with any implementation of those services.

In addition to the previous uses, this project also defines the format descriptions and analyzers for HTML and RDF/XML, since they can take advantage of the already necessary dependencies.

6.2.3 `SFI.BaseFormats`

This project adds several formats and analyzers, but without requiring any additional dependencies. It also supports expressing compound formats, via `ContainerFileFormat`.

6.2.4 `SFI.ExternalFormats`

This project groups all the remaining supported formats that are implemented using external dependencies.

6.2.5 `SFI.MediaAnalysis`

This project stores code for manipulating audiovisual data that requires native support. It mainly adds support for image and audio formats, using GDI+ and NAudio.

6.2.6 `SFI.Windows`

This project adds formats and analyzers implemented solely through the Windows API, using P/Invoke, such as Win32 executable modules, shell links, or cabinet archives.

6.2.7 `SFI.Application`

This is the primary project to use when wishing to run the software using the command line-based API as described in section 6.1, but still in a platform-agnostic way.

The project contains a hierarchy of classes derived from `Inspector`, capable of easily setting up the process of file format inspection. This class is inherited by `ComponentInspector`, which is capable of storing configurable collections of components (section 6.1.1), and itself inherited by `ExtensibleInspector`, adding support for plugins, described in detail by section 6.3.

When intending to use one of these classes, the user shall derive from the one of them covering the user's needs, adding the desired components, and either use the inspector by calling methods on the instance, or through the `Application` class which can be controlled using command-line arguments.

6.2.8 `SFI.ConsoleApp`

This project implements an executable console application for .NET 5.0, providing the components that can be used on the desktop platform.

6.2.9 `SFI.WebApp`

This is an ASP.NET Core Blazor WebAssembly project with a single page for running the application in the browser. It adds only the components usable in the web environment, and loads all required files through file dialogs.

Details about running the web application are provided in section 6.4.

6.2.10 `SFI.SamplePlugin`

Stores additional sample components, for illustrating how they can be added to the application as plugins. The process is explained in greater detail in section 6.3.

6.2.11 `SFI.Tests`

This project contains test classes and methods that cover other parts of the solution. Details about the testing methodology are presented in chapter 7.

6.3 Extender documentation

The application may be extended with plugins, capable of adding new components with arbitrary types, but still configurable through the application as needed. This section describes both how plugins are loaded into the application by the user and how they interact with the application through the .NET API.

6.3.1 Loading plugins

Plugins are stored in directories or ZIP files in the `plugins` directory alongside the main executable in the case of the console applications, or provided only as ZIP files for the web applications. Each plugin has a main .NET assembly, which is looked up by replacing the `.zip` extension with `.dll` in the case of ZIP files, or just appending `.dll` to the directory name in the case of a directory.

The main assembly of a plugin should contain publicly visible and constructible types, compatible with one of the known component collections[15].

When such a type is encountered, dependency injection is used to select the appropriate constructor, providing these services distinguished by their type:

- `Inspector` – the inspector instance that is loading the plugin,
- `TextWriter` – an instance used for logging messages,
- `IDirectoryInfo` – the directory where the main assembly of the plugin is located.

Once an instance of the type is created, it is added into all compatible collections, and thus becomes in use by the application.

Example

The following pieces of codes are examples taken from the `SFI.SamplePlugin` project, showing how to add custom analyzers, formats, and hash algorithms.

[15]The initially defined component collections are the collection of analyzers, implementing `IEntityAnalyzer<T>`, the collection of container formats, implementing `IContainerAnalyzerProvider`, the collection of data formats, implementing `IBinaryFileFormat`, the collection of XML formats, implementing `IXmlDocumentFormat`, the collection of data hash algorithms, implementing `IDataHashAlgorithm`, the collection of file hash algorithms, implementing `IFileHashAlgorithm`, and for the image analyzer, the collection of image hashes, implementing `IObjectHashAlgorithm<Image>`, and pixel hashes, again implementing `IDataHashAlgorithm`. See section 6.1.1 for more details.

Formats. Custom data formats are identified based on whether they implement `IBinaryFileFormat`, either directly or through the base class or other interfaces that inherit it:

```
// BinaryFileFormat implements IBinaryFileFormat
public class UriListFormat : BinaryFileFormat<IReadOnlyList<Uri
    ↪>>
{
    public UriListFormat() : base(246, "text/uri-list", "uris")
    {

    }

    // Returns true if it is likely in this format
    public override bool CheckHeader(ReadOnlySpan<byte> header,
        ↪ bool isBinary, IEncodingDetector? encodingDetector)
    {
        // Binary files can be immediately
        if(isBinary) return false;
        using var reader = new StringReader(Encoding.UTF8.
            ↪GetString(header));
        while(reader.ReadLine() is string line)
        {
            if(Uri.TryCreate(line, UriKind.Absolute, out _))
            {
                return true;
            }
        }
        return false;
    }

    // Parses the file. Exceptions may be thrown from the Uri
        ↪constructor, indicating that the format is incorrect.
    public override async ValueTask<TResult?> Match<TResult,
        ↪TArgs>(Stream stream, MatchContext context,
        ↪ResultFactory<IReadOnlyList<Uri>, TResult, TArgs>
        ↪resultFactory, TArgs args) where TResult : default
    {
        using var reader = new StreamReader(stream);
        var list = new List<Uri>();
        while(await reader.ReadLineAsync() is string line)
        {
            if(line.StartsWith("#")) continue;
            list.Add(new Uri(line, UriKind.Absolute));
        }
        // The resulting entity is returned through the callback
            ↪.
        return await resultFactory(list, args);
    }
}
```

Listing 6.6: An example of a custom data format component for `text/uri-list`

Analyzers. Custom analyzers are identified on the basis of whether they implement `IEntityAnalyzer<T>`, for any type `T`:

```
// EntityAnalyzer implements IEntityAnalyzer
public class UriListAnalyzer : EntityAnalyzer<IReadOnlyList<Uri
   ↪>>
{
    // Recommended async to capture exceptions in the task.
    public override async ValueTask<AnalysisResult> Analyze(
       ↪IReadOnlyList<Uri> entity, AnalysisContext context,
       ↪IEntityAnalyzers analyzers)
    {
        // Construct the linked node appropriately for the
           ↪context.
        var node = GetNode(context);

        // Set its properties to the URIs stored in the list.
        for(int i = 0; i < entity.Count; i++)
        {
            node.Set(Properties.MemberAt, i + 1, entity[i]);
        }

        // Return through AnalysisResult.
        return new(node);
    }
}
```

Listing 6.7: An example of a custom analyzer component for `text/uri-list`

Hash algorithms. Custom data hash algorithms are identified based on whether they implement `IDataHashAlgorithm`:

```
using Murmur;

// BuiltInHash implements IDataHashAlgorithm
public class Murmur32Hash : BuiltInHash<Murmur32>
{
    // Vocabulary item to identify the hash algorithm
    [Uri(Vocabularies.Uri.At)]
    public static readonly IndividualUri Murmur32;

    // Setting up the hash algorithm through the BuiltInHash
       ↪base
    public Murmur32Hash() : base(() => MurmurHash.Create32(),
       ↪Murmur32, "urn:murmur32:", 0x23, null, Services.
       ↪FormattingMethod.Base64)
    {

    }

    // Populate the vocabulary items
    static Murmur32Hash()
    {
        typeof(Murmur32Hash).InitializeUris();
    }
}
```

Listing 6.8: An example of a custom hash algorithm component using the `Murmur.Murmur32` class provided by an external library

These pieces of code are enough to expose the three custom components, and if they are built into an assembly named `SamplePlugin.dll`, it can be placed in the directory `plugins/SamplePlugin/` to be loaded by the application, or stored as a ZIP file in `plugins/SamplePlugin.zip`, and the components will be automatically loaded.

6.3.2 Configurable components

Components can be configured from the application, allowing it to assign their properties to different values, or to specify other components that they can use. The application uses `System.ComponentModel.TypeDescriptor.GetProperties` on the instance to obtain the list of properties. This method uses reflection by default, but can be configured to provide different values per type or even per instance. Properties that are marked non-browsable[16] are excluded from this collection.

Assignable properties. Any writable property on a component can be assigned from the command-line application, as long as the type converter corresponding to the property supports conversion to and from `System.String`[17]. This converter is used for subsequent conversions, as documented in section 6.1.1.

Custom component collections A component may define a property with the `[ComponentCollection]` attribute to indicate that the elements in the collection shall be treated as components, and subsequent types found in plugins compatible with the element type of the collection may be constructed and placed there. The type of the collection must implement `ICollection<T>`, where `T` can be `System.Object` or a more specific type. The `ComponentCollection` attribute also makes it possible to specify the type of the components explicitly, supporting generic type definitions in addition to concrete types.

Example

[16]For example, by adding the `[Browsable(false)]` attribute.

[17]The converter can be changed via the `[TypeConverter]` attribute on the property or its type.

```
// This property will be exposed as an assignable property "int
   ↪-property"
public int IntProperty { get; set; }

// This property will not be exposed because there is no native
   ↪ conversion from string to int[]
public int[] ArrayProperty { get; set; }

// This property will not be exposed due to the Browsable
   ↪attribute
[Browsable(false)]
public string StringProperty { get; set; }

// This property will not be exposed directly, but classes that
   ↪ implement IJsonObjectFormat will be automatically put
   ↪inside the collection, and may be removed through the json
   ↪-format: prefix
[ComponentCollection("json-format")]
public ICollection<IJsonObjectFormat> JsonFormats { get; } =
   ↪new List<IJsonObjectFormat>();
```

Listing 6.9: An example of assignable and non-assignable properties

6.4 Administrator documentation

The software can also be executed as a web application, thanks to the platform-independent nature of the main application implementation.

The web application is stored in the `SFI.WebApp` project, providing a user interface to the command line-based applications, and the necessary implementations of the required services specific to the web environment. It also excludes components incompatible with the web environment, such as those requiring native code or platform-specific APIs.

The web application may be downloaded from GitHub[18] or built from the source code using the following steps:

1. Get the project from the repository, either via `git clone` or by downloading the ZIP.
2. Open the solution file `SFI.sln` in Visual Studio[19]
3. Right-click on the project `SFI.WebApp` and select “Set as Startup Project”.
4. Pressing F5 or selecting “Debug/Start Debugging” will launch the website in the browser using a local web server.
5. Alternatively, right-click the project and select “Publish...”. You can publish using the default `FolderProfile`, which will output the website into the `bin/Release/net5.0/browser-wasm/publish/` directory.

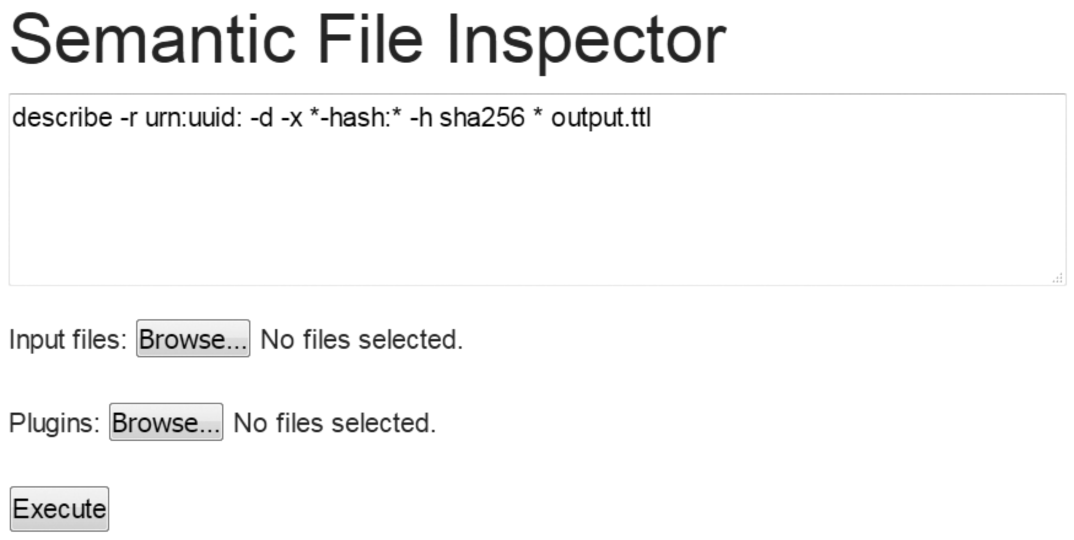
[18]`https://github.com/cermakmarek/SFI/releases/tag/v1.0`

[19]At least Visual Studio 2019, version 16.7, is required.

6. After publishing, the resulting website can be found inside `wwwroot/`.

The website targets ASP.NET Core Blazor WebAssembly, and can therefore be statically hosted on any web server and executed in any browser capable of running WebAssembly. Optionally, the files ending on `.gz` and `.br` can be used by the web server for offering gzip and Brotli-compressed files.

When the web application is loaded, a page similar to the one below is displayed:



```
describe -r urn:uuid: -d -x *-hash:* -h sha256 * output.ttl
```

Figure 6.1: Overview of the web application

The page has the following controls:

- Command prompt – the large text area control is used to enter the arguments to the application, in the same syntax as for the console-based application. There may be multiple lines, in which case the application is executed multiple times for each line.
- Input files – this control is used to enter any input files visible to the application, including described files or SPARQL queries.
- Plugins – this control is used to provide plugins to the application, as individual ZIP archives.
- Execute – launches the application using the provided command-line arguments, as documented in section 6.1. Any output of the application, including the standard output or files, appears below.

7. Tests

This chapter describes the test practices and methodology used for the implementation of the software, ensuring that it behaves according to the design, its functionality is not negatively impaired by new changes to the code, and that it meets the requirements stated in section 3.2.1.

The first section in this chapter introduces *MSTest*, the primary testing framework used by this software, followed by sections describing the various kinds of performed tests in detail.

7.1 MSTest

MSTest is a Microsoft-provided testing framework and SDK, which is supported by Visual Studio by default. As such, it is very straightforward to use for .NET projects opened in Visual Studio, requires no configuration, and its controls and results are integrated into the UI.

Performing tests only requires creating a new *MSTest* project, and using attributes in the `Microsoft.VisualStudio.TestTools.UnitTesting` namespace to mark classes and methods to be automatically executed as part of test suites. The `[TestClass]` attribute is used to mark any class that contains test methods, which are themselves marked by `[TestMethod]`, and will be automatically executed during testing. Methods with parameters can be provided with test values using the `[DataRow]` attribute. The attributes can be used as in the following example, containing two test methods, one of them with parameters:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[TestClass]
public class TestClass
{
    [TestMethod]
    public void TestMethod()
    {
        Assert.Equals(1 + 1, 2);
    }

    [TestMethod]
    [DataRow(-1)]
    [DataRow(1)]
    public void TestMethodWithParameter(int arg)
    {
        Assert.Equals(arg * arg, 1);
    }
}
```

Listing 7.1: Example usage of MSTest

The `Assert` class is used to assert conditions within the tests.

The framework is capable of running individual test methods, all test methods in a particular test class, or all test methods in the solution. For any failing or inconclusive test, it displays the particular test method and input that led to that result in the overall report.

7.2 Unit testing

An important part of software testing is ensuring the functionality of small individual pieces of code, i.e. units. In practice, units are usually individual public methods, ideally pure, i.e. with predictable results and no side effects.

Due to the component-oriented nature of the whole system, and the viability of other tests with higher coverage, the majority of the code is covered by other types of tests, while unit tests are left only for non-component code that does not interact with other parts of the code on its own. These tests are located in the `UnitTests` directory, and cover about 69 % of non-component classes in the main project.

7.3 Component testing

As the software follows a component-oriented architecture, it is viable to test individual components and how they interact with the system.

The tests for components are located in the `ComponentTests` directory, named based on the component they are for. These tests do not necessarily have to cover every component in the system, as the majority of formats and analyzers is better tested using output testing, but the core components like `DataAnalyzer` and `FileAnalyzer` can be tested specifically, to identify issues that could affect other tests.

MSTest is used for these tests as well.

7.3.1 Mocks

Mocking is the creation of objects that mimic real objects and simulate their behaviour but without bringing in the complexities associated with these objects.

There are several types that are necessary to be used mocks for, as their implementations are very complex or cannot be easily used for test conditions:

`ILinkedNode` The real-world implementation of `ILinkedNode`, as defined in section 5.4.7, constructs RDF triples when operated on and sends them to an RDF serializer or graph, but it does not offer any capabilities of reading the data that has been written through it. The mock implementation, `StorageLinkedNode`, instead stores an associative collection mapping a predicate URI to a set of object URIs associated with the subject represented by the `ILinkedNode`. Thanks to this, the objects can be easily retrieved from the collection and compared to the expected values, testing that the component stores the correct properties to the node.

`IEntityAnalyzers` This interface serves to analyze an object produced by another component using an implementation-defined collection of analyzers, as defined in section 5.4.8, but for the purpose of testing individual components, no concrete analyzers should be used. The mock implementation, `AnalyzedObjectCollection`, only stores the analyzed entity in its internal collection, so that it can be retrieved later and compared with the expected object. This can be used to test that a component is capable of deriving the correct objects from another object, and using the `IEntityAnalyzers` instance for the analysis of these entities.

7.4 Output testing

A large part of testing the functionality of the software as a whole is testing the file format analysis process by comparing its RDF output across updates to look for regressions that may affect the result. These tests also have additional uses beyond maintaining correct behaviour – their output could serve as a part of the documentation, as examples of how the software works and how the input files are described in RDF, which properties are covered and which are not.

This testing process is rather simple: the core part of the software is repeatedly executed as if run from the command-line application, with the default components and settings. Each time, it is provided with a different input – the to-be analyzed file – and the output is compared, as RDF, to the stored output of the previous analysis of the same input file, which is a part of the repository. The outputs are compared as RDF graphs, as there could be differences in the textual form due to variations in the process caused by concurrency, or changes to the serializers, but still unaffecting the actual triples. The concrete graph-matching algorithm is provided by dotNetRDF as an implementation detail, but it does not have to be complex, since the output can be configured not to use blank nodes at all and use other means of stable identification of data, such as hashes. For this reason, just comparing the graphs as sets of triples is, in most cases, enough.

If the graphs do not match, this is reported as a test failure, even in situations when triples are only added to the original graph to form the new one, for example as a result of support for new properties or file formats. This encourages updating the stored output to accommodate even for additive changes which do not negatively impair the functionality of the software and serves as a practical documentation of the impact of such changes.

The standard graphs used for comparing are stored in the directory `ExpectedDescriptions`, serialized as Turtle files. If the input file does not have a previously stored description, the current one is saved into `NewDescriptions`, otherwise if the descriptions do not match, the new one is saved into `NotMatchedDescriptions`. Files downloaded from the web are cached in `Cached`. In all these directories, the files are identified using the version 5 UUID created from the URIs identifying the original files.

For performing these tests, *MSTest* is used as well, in the class `GraphTests`. The tests are separated according to the source of the file samples.

7.4.1 telparia.com

telparia.com[1] hosts a large and still growing database of over 14000 files, grouped into more than 1700 formats.

The file formats are organized into general categories, such as archives, documents, executables, images etc., containing the directories for individual formats, each storing samples of that format. Neither the general category directories nor the specific format directories follow any standardized file format naming scheme, such as MIME; the names are arbitrary, but descriptive for humans.

In this database, 2016 relevant files were identified, covering 38 of the supported 41 formats and their corresponding analyzers, equating to 92.7% coverage.

The only formats not covered by this database are RDF/XML and `.lnk` and `.url` files. Samples for these formats can be found in the `Samples` directory.

7.5 Manual testing

This section describes the tests that can be performed manually, ensuring that the software behaves according to the functional requirements specified in section 3.2.1. It is assumed the directory `SFI.Tests/Samples`[2] is located in the working directory.

Functional requirements under FR4, specifying the support for adding new components, can be tested by following the extender documentation in section 6.3.

7.5.1 Obtaining a description of a file

1. Launch the application with the parameters `describe -d -r urn:uuid: -q -x *-hash:* -h sha256 Samples/zip/example.zip -`.
2. The application should produce a Turtle-encoded RDF graph, equivalent to the attachment in section A.2.

By inspecting the output, these requirements can be verified:

- FR1.1 – Common file properties

  Properties of each file are present in the output, such as `nfo:fileName`, `nfo:fileSize`, and `nfo:fileLastModified`.

- FR1.2 – Recognizing known formats

  The formats PNG and ZIP are recognized from the input[3].

- FR1.3 – Properties of known formats

  The recognized formats are identified using their media types `image/png` and `application/zip`. The extension is reflected in the label.

[1]`https://telparia.com/fileFormatSamples/`

[2]`https://github.com/cermakmarek/SFI/tree/master/SFI.Tests/Samples`

[3]The ability to recognize other formats may be assessed by output testing, described in section 7.4.

- FR1.4 – Analyzing recognized files

  The PNG image, represented by the resource `<ni:///sha-256;...?ct=image/png>`, has been analyzed and described using many image-related properties, such as `nfo:width`, `nfo:height`, and `nfo:colorDepth`.

- FR1.5 – Analyzing unrecognized binary files

  The unrecognized binary file `unk-binary.dat` has been described using the implied media type `application/x.sig.head`, storing its signature.

- FR1.6 – Analyzing unrecognized textual files

  The unrecognized text file `unk-textual.txt` has been described using its first line, represented as `<ni:///sha-256;...?ct=text/plain#line=,1>`.

- FR1.7 – Computing hashes

  SHA-256 hashes were computed from the 3 encountered files, as `<urn:sha256:...>` nodes, and linked to the respective content nodes.

- FR1.8 – Traversing nested file systems

  The file is a ZIP archive which stores an inner file system, which is reflected by the hierarchy starting from `<ni:///sha-256;...?ct=application/zip#/>`.

- FR2.1 – Reuse of existing ontologies

  Almost all used properties and classes are from pre-existing vocabularies, with the exception of `at:digest`, `at:pathObject`, and `at:extensionObject`.

- FR2.2 – Use of hierarchical URIs

  The URIs produced for the files inside the ZIP archive follow a hierarchy, starting from `ni:///sha-256;...?ct=application/zip#/`, followed by the path to each file.

- FR2.3 – Elimination of blank nodes

  No blank nodes were produced in the output, thanks to the choice of SHA-256 as the primary hash algorithm to identify individual pieces of content. If there were any nodes that could be identified this way, they would use the root URI prefix provided by `-r`.

- FR2.4 – Separate nodes for the file, its content, and the media object serialized it in

  Nodes such as `<ni:///sha-256;...?ct=application/zip#/dir/example.png>`, `<ni:///sha-256;...?ct=application/octet-stream>`, and `<ni:///sha-256;...?ct=image/png>` are distinct from each other, and represent the files at various levels of interpretation.

7.5.2 Searching in files

1. Launch the application with the parameters `search -s - -q -x *-hash: * -h sha256 Samples/zip/example.zip -`.

2. Provide the following SPARQL query through the standard input:

```
PREFIX nfo: <http://www.semanticdesktop.org/ontologies
   ↪/2007/03/22/nfo#>
SELECT * WHERE {
  ?image nfo:width ?width .
  ?image nfo:height ?height .
}
```

Listing 7.2: A test SPARQL query to obtain the properties of images

3. The application should produce these SPARQL XML results:

```
<?xml version="1.0" encoding="utf-8"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="image" />
    <variable name="height" />
    <variable name="width" />
  </head>
  <results>
    <result>
      <binding name="image">
        <uri>ni:///sha-256;FrAhAufBERWjweVgm65W-
           ↪LxtbrApgClbV-SbFS36IiY?ct=image/png</uri>
      </binding>
      <binding name="height">
        <literal datatype="http://www.w3.org/2001/XMLSchema
           ↪#integer">44</literal>
      </binding>
      <binding name="width">
        <literal datatype="http://www.w3.org/2001/XMLSchema
           ↪#integer">144</literal>
      </binding>
    </result>
  </results>
</sparql>
```

Listing 7.3: Test SPARQL XML results with the properties of images

The correctness of the result may be verified by comparing it to the graph attached in section A.2.

This tests FR1.9 – Using SPARQL for search.

7.5.3 Extracting sub-files

1. Launch the application with the parameters `describe -s - -q -x *-hash :* -h sha256 Samples/zip/example.zip NUL`.

2. Provide the following SPARQL query through the standard input:

```
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX schema: <http://schema.org/>
SELECT * WHERE {
  ?node dcterms:hasFormat/schema:encodingFormat <https://
     ↪w3id.org/uri4uri/mime/application/x.sig.head> .
  BIND("-" AS ?path_format)
}
```

Listing 7.4: A test SPARQL query to extract a file in a particular format

3. The application should extract the file `unk-binary.dat` from the archive and write it to the standard output, starting with the text `HEAD`.

This tests FR1.10 – Using SPARQL for extraction.

8. Evaluation

This chapter shows the usage of the application in practice on a large amount of data, evaluates its results, shows various interesting examples and statistics, and assesses the meeting of goals.

8.1 Preparation

The application was used to describe in detail the Windows 10[1] and Windows 11[2] installation media, downloaded as ISO files from Microsoft services.

The files were processed with the command-line parameters `-x *-hash:* -i *-hash:xxh64 -i image-hash:* -h xxh64 -r urn:uuid:`, using the xxHash-64 algorithm as the primary data hash.

The following table contains some basic information about the input files and the output description serialization produced by the application:

Input	**Windows 10**	**Windows 11**
File name	`Win10US_x64.iso`	`Win11_22H2_English-International_x64v1.iso`
File size	4.455 GiB	5.167 GiB
Downloaded on	2023-02-09	2023-02-09
SHA-1 hash	`b59cb70a1349cdee-5c12ddd94558483e-3938c216`	`8b3c9f859e3273ff-ac425179e501eee1-e4f9db95`
Output	**Windows 10**	**Windows 11**
Turtle size	17.579 MiB	17.864 MiB
Turtle size (GZip)[3]	1.469 MiB	1.503 MiB

Table 8.1: Properties of the input and ouput of the application

8.2 Profiling and statistics

As a next step, the output graphs were loaded into a Virtuoso instance, to simplify browsing and querying[4].

The following table contains triple statistics for the two graphs:

[1] `https://www.microsoft.com/software-download/windows10`

[2] `https://www.microsoft.com/software-download/windows11`

[3] This corresponds to about 0.03 % of the input size.

[4] The results of the queries can be replicated here: `https://cermakmarek.github.io/sfi/example-queries.html`.

Number of	Windows 10	Windows 11
Triples	216138	220067
Distinct triples[5]	140839	140921
Unique triples[6]	98516	98598
Subjects	29529	29380
Unique subjects[7]	17752	17603

Table 8.2: Counts of entities in the output graphs of the application

Since the `-d` parameter was not specified, the input files were described as unique entities with randomly-generated UUIDs, identified using `urn:uuid:21101c75-83e7-4104-b7c4-f53fe44eb970` and `urn:uuid:53e226e1-3a50-482a-98c3-15463643c5d0`[8], for Windows 10 and 11 respectively. The triples were loaded into two named graphs identified by the same URIs.

The benefit of using RDF to describe the metadata becomes apparent from the very first query that can be performed on the graphs, to count the number of times a particular property-value pair appears on distinct resources, using the following SPARQL query:

```
SELECT ?source ?property ?object ((COUNT(DISTINCT ?item)) AS ?
   ↪count)
WHERE {
  VALUES ?graph {
    id:65948b27-93be-4ace-9091-f2272525a6cd
    id:b45785a7-e8bf-4428-8996-14caec7eaaed
  }

  GRAPH ?graph {
    ?graph nfo:fileName ?source .
    ?item ?property ?object .
  }
}
GROUP BY ?source ?property ?object
HAVING (COUNT(DISTINCT ?item) > 4)
ORDER BY DESC(COUNT(DISTINCT ?item)) ASC(?source)
```

Listing 8.1: A SPARQL query that counts the number of distinct subjects for all predicate-object pairs

With this single query, many profiling information can be derived from the two graphs:

[5]Some triples may be duplicated on output in the non-buffered mode, such as triples for duplicate files or `file:` path hierarchies.

[6]There were 42323 triples shared in both graphs.

[7]There were 11777 subjects shared in both graphs.

[8]In later queries, the prefix `id:` is defined as `<urn:uuid:>`.

Number of	Windows 10	Windows 11
All files[9]	5706	5717
Directories[10]	781	776
Recognized media[11]	4732	4706
Binary content[12]	2376	2382
Text content[13]	1329	1301
XML content[14]	1099	1082
Executables[15]	656	668
Images[16]	628	623
Certificates[17]	320	351
Programs from Microsoft[18]	606	565

Table 8.3: Counts of various occurrences within the data

Another general-purpose query could be made to obtain the range of values and average value of properties:

```
SELECT ?source ?property (MIN(?value) AS ?min) (AVG(IF(
  ↪isNumeric(?value),?value,"INF"^^xsd:double)) AS ?avg) (MAX
  ↪(?value) AS ?max)
WHERE {
  VALUES ?graph {
    id:65948b27-93be-4ace-9091-f2272525a6cd
    id:b45785a7-e8bf-4428-8996-14caec7eaaed
  }

  GRAPH ?graph {
    ?graph nfo:fileName ?source .
    ?item ?property ?value .
  }
}
ORDER BY ?property ?source
```

Listing 8.2: A SPARQL query that produces the minimum, maximum value of properties, and average value for numeric objects of properties

[9]a nfo:FileDataObject
[10]a nfo:Folder
[11]a schema:MediaObject
[12]a cnt:ContentAsBase64
[13]a cnt:ContentAsText
[14]a cnt:ContentAsXML
[15]a schema:SoftwareApplication
[16]a schema:ImageObject
[17]a cert:X509Certificate
[18]dcterms:creator "Microsoft Corporation"@en-us

Data	Windows 10	Windows 11
Average file size[19]	1.9 MiB	2.19 MiB
Average media dimensions[20]	60x33	73x36
Maximum media dimensions[21]	1366x800	1025x768
Earliest file creation date[22]	2022-09-08T08:07:16Z	2022-09-25T03:41:03.819Z
Earliest file modification date[23]	2019-04-18T18:43:00Z	2022-05-04T17:34:00Z
Earliest certificate expiration date[24]	2010-01-22T22:34:55+01:00	2013-01-10T21:32:25+01:00

Table 8.4: Properties of various occurrences within the data

For all properties, counting the distinct values may also be informative:

```
SELECT ?source ?property ((COUNT(DISTINCT ?object)) AS ?count)
WHERE {
  VALUES ?graph {
    id:65948b27-93be-4ace-9091-f2272525a6cd
    id:b45785a7-e8bf-4428-8996-14caec7eaaed
  }

  GRAPH ?graph {
    ?graph nfo:fileName ?source .
    [] ?property ?object .
  }
}
GROUP BY ?source ?property
ORDER BY DESC(COUNT(DISTINCT ?object)) ASC(?source)
```

Listing 8.3: A SPARQL query that counts the number of distinct objects for each property

Number of	Windows 10	Windows 11
Media types[25]	54	52
File names[26]	3524	3454

[19]`nfo:fileSize`, including the whole input file.
[20]`nfo:widthxnfo:height`
[21]`nfo:widthxnfo:height`
[22]`nfo:fileCreated`
[23]`nfo:fileLastModified`
[24]`sec:expiration`
[25]`schema:encodingFormat`
[26]`nfo:fileName`

Hashes[27]	5089	5045

Table 8.5: Properties of various occurrences within the data

8.2.1 Observations

Based on the information collected earlier from general queries, we can make several observations and formulate questions that might be of interest:

- There are two media types missing from the Windows 11 installation – what are they?
- There are 28 text files missing from the Windows 11 installation compared to Windows 10 – which of them are they?
- The number of programs from Microsoft is smaller than the number of executables – which programs were not made by Microsoft?
- The total number of files is larger than the total number of text and binary content – which files are duplicates?

Missing entities

The questions about missing files and media types might be answered both using a single configurable SPARQL query:

```
SELECT ?source ?property ?object
WHERE {
  VALUES (?graph_a ?graph_b) {
    (id:65948b27-93be-4ace-9091-f2272525a6cd id:b45785a7-e8bf
       ↪-4428-8996-14caec7eaaed)
    (id:b45785a7-e8bf-4428-8996-14caec7eaaed id:65948b27-93be-4
       ↪ace-9091-f2272525a6cd)
  }
  GRAPH ?graph_a {
    ?graph_a nfo:fileName ?source .
    VALUES ?property {
      nie:interpretedAs
      schema:encodingFormat
    }
    [] ?property ?object .
    FILTER NOT EXISTS {
      GRAPH ?graph_b {
        [] ?property ?object .
      }
    }
```

[27]at:digest

```
  }
}
ORDER BY ?property ?object
```

Listing 8.4: A SPARQL query that lists the objects of properties `nie:interpretedAs` and `schema:encodingFormat` that are missing from either graphs

Source	**Property**	**Object**
"Win10US_x64.iso"	schema: encodingFormat	mime:application/x. ns.switches+xml
"Win10US_x64.iso"	schema: encodingFormat	mime:application/x. sig.crim0
"Win10US_x64.iso"	schema: encodingFormat	mime:application/x. sig.crimt
"Win11_22H2_ English- International_ x64v1.iso"	schema: encodingFormat	mime:application/x. sig.crimx

Table 8.6: The first 4 results of the query in Listing 8.4

This partially answers the questions already – one of the differences in media types was caused only by misinterpreting the files with the `CRIM` signature, which got mixed with the following byte, errorneously interpreted as an ASCII character. The other missing media type, `application/x.ns.switches+xml`, was implied from the `<Switches>` root element of the XML document[28], stored as an embedded resource inside `/sources/generaltel.dll`[29], as can be observed by traversing alongside the relevant property paths. This DLL, described as "General Telemetry", does have its equivalent in the Windows 11 graph[30], but it lacks the corresponding resource.

This query does not sufficiently answer the question of missing text files however, because the number of text files differing between the two graphs is too large to compare. This can be improved by a more precise query:

```
SELECT ?source ?file
WHERE {
  VALUES (?graph_a ?graph_b) {
    (id:65948b27-93be-4ace-9091-f2272525a6cd id:b45785a7-e8bf
       ↪-4428-8996-14caec7eaaed)
  }
  GRAPH ?graph_a {
    ?graph_a nfo:fileName ?source .
    ?file nie:interpretedAs ?object .
```

[28] `ni:///mh;4ucCCAW6xSlj0UwB?ct=application/xml`
[29] `ni:///mh;4ucCCBf7SPjkKnI0?ct=application/x-udf#/sources/generaltel.dll`
[30] `ni:///mh;4ucCCA1uFg5qaoIm?ct=application/x-udf#/sources/generaltel.dll`

```
    ?file at:pathObject ?path .
    ?object a cnt:ContentAsText.
    FILTER NOT EXISTS {
      GRAPH ?graph_b {
        { [] nie:interpretedAs ?object . } UNION { [] at:
            ↪pathObject ?path . }
      }
    }
  }
}
ORDER BY ?file
```
Listing 8.5: A SPARQL query that lists the files with textual content whose path and content are missing from the other graph

This yields 86 files, which were neither updated nor moved unchanged in the Windows 11 installation. After manual inspection, they were attributed to the following differences:

- `Microsoft-Windows-NetFx3-OnDemand-Package~31bf3856ad364e35~amd64~en-US~.cab`[31] missing from `/sources/sxs/`.
- `/sources/SetupDU_159931.spdx.json`[32] updated and renamed to `SetupDU_161400.spdx.json`[33].
- `/sources/en-us/`[34] renamed to `en-gb/`[35].
- Files removed from `/sources/`[36].
- Several `.manifest` files in Cabinet archives[37] either moved or deleted.
- Embedded resources inside two `PresentationHostDll.dll.mui` files[38], both of which are missing.
- `/Html/FEEDBACKTOOL.XSL`[39] inside `/sources/devinv.dll`[40], whose Windows 11 version[41] is missing the resource.

31 ni:///mh;4ucCCBf7SPjkKnI0?ct=application/x-udf#/sources/sxs/Microsoft-Windows-NetFx3-OnDemand-Package~31bf3856ad364e35~amd64~en-US~.cab

32 ni:///mh;4ucCCBf7SPjkKnI0?ct=application/x-udf#/sources/SetupDU_159931.spdx.json

33 ni:///mh;4ucCCA1uFg5qaoIm?ct=application/x-udf#/sources/SetupDU_161400.spdx.json

34 ni:///mh;4ucCCBf7SPjkKnI0?ct=application/x-udf#/sources/en-us/

35 ni:///mh;4ucCCA1uFg5qaoIm?ct=application/x-udf#/sources/en-gb/

36 ni:///mh;4ucCCBf7SPjkKnI0?ct=application/x-udf#/sources/

37 ni:///mh;4ucCCIu2-uLxmASs?ct=application/vnd.ms-cab-compressed, ni:///mh;4ucCCMZZqDfMq1oq?ct=application/vnd.ms-cab-compressed, ni:///mh;4ucCCKO_p5j6r9So?ct=application/vnd.ms-cab-compressed

38 ni:///mh;4ucCCHYZqj8AoPDb?ct=application/vnd.microsoft.portable-executable, ni:///mh;4ucCCMqLvAPoJ-7d?ct=application/vnd.microsoft.portable-executable

39 ni:///mh;4ucCCPNMKWBmKcjm?ct=application/vnd.microsoft.portable-executable#/Html/FEEDBACKTOOL.XSL

40 ni:///mh;4ucCCPNMKWBmKcjm?ct=application/vnd.microsoft.portable-executable

41 ni:///mh;4ucCCDHCeASx-2ps?ct=application/vnd.microsoft.portable-executable

Non-Microsoft executables

As could be seen from the profiling data, not all executable files were made by `"Microsoft Corporation"@en-us`. It is, however, possible that there were other variations of the literal:

```
SELECT DISTINCT ?exe ?creator
WHERE {
  VALUES ?graph {
    id:65948b27-93be-4ace-9091-f2272525a6cd
    id:b45785a7-e8bf-4428-8996-14caec7eaaed
  }
  GRAPH ?graph {
    ?exe a schema:SoftwareApplication .
    FILTER NOT EXISTS {
      VALUES ?microsoft {
        "Microsoft Corporation"@en-us
        "Microsoft Corporation"@en-gb
        "Microsoft Corporation"
      }
      ?exe dcterms:creator ?microsoft .
    }
    OPTIONAL {
      ?exe dcterms:creator ?creator .
    }
  }
}
```

Listing 8.6: A SPARQL query that identifies applications not made by Microsoft

This yields 4 occurrences of `PresentationCFFRasterizer.dll`[42] from Adobe Systems Incorporated, a boot-related COM executable `etfsboot.com`[43], and everal other executables lacking any creator or description: `imagelib.dll`[44], `setupdiag.exe`[45], `hwreqchk.dll`[46], `compatappraiserresources.dll`[47], and `compatappraiserresources.dll.mui`[48].

Duplicates

Finding duplicates is just a matter of grouping files based on their content, since pieces of content are identified solely by the stored byte sequence:

[42] `ni:///mh;4ucCCMkU0Sp24U-5?ct=application/vnd.microsoft.portable-executable`, `ni:///mh;4ucCCIEdWMtPZF5Y?ct=application/vnd.microsoft.portable-executable`, `ni:///mh;4ucCCPS7VcMZPrUH?ct=application/vnd.microsoft.portable-executable`, `ni:///mh;4ucCCCyg7xOUxZmS?ct=application/vnd.microsoft.portable-executable`

[43] `ni:///mh;4ucCCD8V4mkHRiiz?ct=application/x-dosexec`

[44] `ni:///mh;4ucCCGc1nrs3Y8mc?ct=application/vnd.microsoft.portable-executable`

[45] `ni:///mh;4ucCCOeJ-iXW7VyP?ct=application/vnd.microsoft.portable-executable`, `ni:///mh;4ucCCBf7SPjkKnI0?ct=application/x-udf#/sources/setupdiag.exe`

[46] `ni:///mh;4ucCCDppzpHBiB1C?ct=application/vnd.microsoft.portable-executable`

[47] `ni:///mh;4ucCCDpURMyTF9Kg?ct=application/vnd.microsoft.portable-executable`

[48] `ni:///mh;4ucCCBVZGsPDxend?ct=application/vnd.microsoft.portable-executable`

```
SELECT ?source (SUM(?count) AS ?total_count) (SUM(?size) AS ?
   ↪size)
WHERE {
  {
    SELECT ?source ((COUNT(?file)-1) AS ?count) ((SUM(?size)-
       ↪SAMPLE(?size)) AS ?size)
    WHERE {
      VALUES ?graph {
        id:65948b27-93be-4ace-9091-f2272525a6cd
        id:b45785a7-e8bf-4428-8996-14caec7eaaed
      }

      GRAPH ?graph {
        ?graph nfo:fileName ?source .
        ?file nie:interpretedAs ?content .
        { ?content a cnt:ContentAsText . } UNION { ?content a
           ↪cnt:ContentAsBase64 . }
        ?file nfo:fileSize ?size .
      }
    }
    GROUP BY ?source ?content
    HAVING (COUNT(?file) > 1)
  }
}
GROUP BY ?source
```
Listing 8.7: A SPARQL query that counts the total number and size of duplicated files

This query results in 81.8 MiB of duplicate data over 1234 files for Windows 10, and 56.4 MiB over 1274 files for Windows 11.

Multi-typed files

By navigating to the main content entity for Windows 10[49] and Windows 11[50], it can be observed that the files are in two distinct formats simultaneously, ISO and UDF, and as such contain two separate file systems, depending on what kind format is requested to open. Furthermore, the Windows 11 installation ISO contains a `README.TXT`[51] file with the text “This disc contains a ”UDF” file system and requires an operating system”, which is revealed to applications that do not support UDF.

It is possible to find other similar files that have content that can be interpreted in multiple formats:

```
SELECT ?source ?content
WHERE {
  VALUES ?graph {
```

49 `ni:///mh;4ucCCBf7SPjkKnIO?ct=application/octet-stream`
50 `ni:///mh;4ucCCA1uFg5qaoIm?ct=application/octet-stream`
51 `ni:///mh;4ucCCN40EXLPpT9m?ct=text/plain`

```
    id:65948b27-93be-4ace-9091-f2272525a6cd
    id:b45785a7-e8bf-4428-8996-14caec7eaaed
  }
  GRAPH ?graph {
    ?graph nfo:fileName ?source .
    ?content dcterms:hasFormat ?format .
    FILTER NOT EXISTS {
      # Ignore certificates, interpreted from executables
      ?format a cert:X509Certificate .
    }
  }
}
GROUP BY ?source ?content
HAVING (COUNT(?format) > 1)
ORDER BY DESC(COUNT(?format)) ASC(?source)
```

Listing 8.8: A SPARQL query that retrieves all pieces of content with multiple formats

Aside from the two main files, the only files found by the query are six HTML files, interpreted as both HTML and XML due to the similarity of the formats.

8.2.2 Benefits of RDF and SPARQL

As can be seen from the examples above, using RDF as the representation of metadata and SPARQL as a querying language has several benefits:

- The data can be stored in a triple store, allowing one to browse, identify, and link individual resources using any features supported by the triple store.
- It is possible to write completely domain-agnostic queries that still provide useful statistics, such as obtaining the number of distinct objects.
- More varied queries can be written easily, using only the relevant vocabulary, and operating on multiple graphs at once.
- Differences in the data can be observed both on a large scale using specialized queries and on a small scale by navigating to individual resources and exploring alongside property paths.
- All of the above is possible while working with graphs completely detached from the original input files, taking less than 1 % of the original size.

8.3 Meeting goals

The primary goal of the application was stated to be capable of generating descriptions of input files by identifying the formats they are in and extracting the relevant metadata. As is evident from the above process, this goal was achieved completely – the known formats are recognized in files, even multiple formats per file, and they are analyzed to the depth, with metadata collected from every piece of content and media that is encountered.

Several potential uses of the application were also outlined:

Exploration of file systems

Various kinds of file systems can be browsed and described by the application, and the collective description allows for a range of interesting queries.

File input validation

While the intention of the application was not to collect data, only metadata, and as such not all methods of validation may be performed through the RDF description, it is nonetheless able to store many relevant properties of the data and may thus be of use in at least some file validation situations.

Linking files from different data sources

While excluded from the specific evaluation scenario provided above, a wide variety of hashes from files and data may be computed, allowing to find records of them on external sites, such as those specialized in providing DLL files. Despite the use of standardized URI schemes to produce the identifiers, identifiers other than hashes have limited use for this purpose, since most file hosting sites use their own proprietary way of identification.

Extraction of relevant files based on specific criteria

It is possible to write extraction queries similar to the one shown above that identify matching files using the `?node` variable, or such a query could be constructed from a given list of URIs for each entity.

The architecture of the application does not, however, allow one to extract all entities potentially matched by too complex queries requiring links to other entities in a single step, as the links may have already been purged from memory in the non-buffered mode, or they may appear in the data too late, when an entity has already been processed. This is, however, still possible to achieve in two steps, i.e. to prepare a list of stable URIs from the RDF graph to identify entities for extraction, and to provide that list via a SPARQL query that selects the individual nodes[52].

[52]This process is relatively straightforward, as is shown in section 6.1.2.

Conclusion

Working on this software has been a long and interesting journey, from starting with a simple premise – to describe every file in RDF – and turning it into something that could be considered a full-fledged framework or suite, able to be used on its own as an extensible stand-alone application, or as a component incorporated in a complex solution.

There were many complications that needed to be overcome in this process, from designing the whole system and architecture and representing it in C#, to facing the peculiarities of URI parsing in .NET, finding various bugs in dotNetRDF and working with its authors to resolve them, rescuing uri4uri to preserve and enhance media type descriptions, and learning about the more obscure formats and finding out how to use the varied libraries and APIs able to work with them. However, overcoming each of these steps is what led to additional knowledge and insight, invaluable in getting to the end of the journey.

While this is the end of the thesis, the real journey may never end, as more and more formats and file encoding schemes get invented, and any *perfect* application of this nature would have to be able to understand them all, to claim this attribute.

Regardless of this practical impossibility to reach perfection, there are still many other areas that could be improved – the ability to describe all data in RDF and not just metadata, a more advanced SPARQL engine that could evaluate the queries alongside the inspection process, more varied output with more properties from specialized vocabularies, additional useful heuristics for profiling unrecognized textual and binary files, perhaps incorporating latest AI-based technologies such as GPT, a user-friendly front-end to load the input and present and analyze the output, or a more detailed and controllable plugin system.

There are a lot of features that could be added, some of which worthy of theses on their own, but it is a great wish and hope of the author that this is not a task solely for himself, and that thanks to the extensibility of the application in both directions, others could improve it in any number of ways they see fit, not just for the benefit of the potential users of the application, but for the benefit of the whole RDF and Linked Data community, whose respect and admiration for was what fueled this project from its inception.

Bibliography

[1] David Wood, Markus Lanthaler, and Richard Cyganiak. RDF 1.1 Concepts and Abstract Syntax. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[2] Steven Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C recommendation, W3C, March 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[3] Dimitris Kontokostas and Holger Knublauch. Shapes Constraint Language (SHACL). W3C recommendation, W3C, July 2017. https://www.w3.org/TR/2017/REC-shacl-20170720/.

[4] Gavin Carothers and Eric Prud'hommeaux. RDF 1.1 Turtle. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-turtle-20140225/.

[5] Boris Motik, Bijan Parsia, and Peter Patel-Schneider. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[6] ISO Central Secretary. Information technology — Universal coded character set (UCS) identifiers. Standard ISO/IEC 10646:2020, International Organization for Standardization, Geneva, CH, December 2020.

[7] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180, October 2005.

[8] Jean Paoli, Michael Sperberg-McQueen, Eve Maler, François Yergeau, and Tim Bray. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, W3C, November 2008. https://www.w3.org/TR/2008/REC-xml-20081126/.

[9] Steven Pemberton. XHTML™1.0 The Extensible HyperText Markup Language (Second Edition). WD not longer in development, W3C, March 2018. https://www.w3.org/TR/2018/SPSD-xhtml1-20180327/.

[10] Bogdan Brinza, Dirk Schulze, Amelia Bellamy-Royds, David Storey, Eric Willigers, and Chris Lilley. Scalable Vector Graphics (SVG) 2. Candidate recommendation, W3C, October 2018. https://www.w3.org/TR/2018/CR-SVG2-20181004/.

[11] Marc Hadley, Anish Karmarkar, Martin Gudgin, Noah Mendelsohn, Jean-Jacques Moreau, Yves Lafon, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C recommendation, W3C, April 2007. https://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[12] Zhi Wei Shuang and Daniel Burnett. Speech Synthesis Markup Language (SSML) Version 1.1. W3C recommendation, W3C, September 2010. https://www.w3.org/TR/2010/REC-speech-synthesis11-20100907/.

[13] Guus Schreiber and Fabien Gandon. RDF 1.1 XML Syntax. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/.

[14] L. Peter Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996.

[15] Ned Freed, Dr. John C. Klensin, and Tony Hansen. Media Type Specifications and Registration Procedures. RFC 6838, January 2013.

[16] Ned Freed and Dr. Nathaniel S. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046, November 1996.

[17] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[18] Dr. John C. Klensin. Simple Mail Transfer Protocol. RFC 5321, October 2008.

[19] Larry M Masinter. The ”data” URL scheme. RFC 2397, August 1998.

[20] Henry Thompson, Tim Bray, Andrew Layman, Richard Tobin, and Dave Hollander. Namespaces in XML 1.0 (Third Edition). W3C recommendation, W3C, December 2009. https://www.w3.org/TR/2009/REC-xml-names-20091208/.

[21] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, January 2005.

[22] Murray Maloney, David Beech, Noah Mendelsohn, Henry Thompson, Sandy Gao, and Michael Sperberg-McQueen. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. W3C recommendation, W3C, April 2012. https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/.

[23] Eamonn Neylon, Tony Hammond, Herbert Van de Sompel, and Dr. Stuart Weibel. The ”info” URI Scheme for Information Assets with Identifiers in Public Namespaces. RFC 4452, April 2006.

[24] Tim Berners-Lee, Larry M Masinter, and Mark P. McCahill. Uniform Resource Locators (URL). RFC 1738, December 1994.

[25] File Transfer Protocol. RFC 959, October 1985.

[26] Matthew Kerwin. The ”file” URI Scheme. RFC 8089, February 2017.

[27] Sandro Hawke and Tim Kindberg. The ’tag’ URI Scheme. RFC 4151, October 2005.

[28] Peter Saint-Andre and Dr. John C. Klensin. Uniform Resource Names (URNs). RFC 8141, April 2017.

[29] Martin J. Dürst and Michel Suignard. Internationalized Resource Identifiers (IRIs). RFC 3987, January 2005.

[30] Stephen Farrell, Dirk Kutscher, Christian Dannewitz, Börje Ohlman, Ari Keränen, and Phillip Hallam-Baker. Naming Things with Hashes. RFC 6920, April 2013.

[31] Eran Hammer-Lahav and Mark Nottingham. Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785, April 2010.

[32] ISO Central Secretary. Information processing — Text and office systems — Standard Generalized Markup Language (SGML). Standard ISO 8879:1986, International Organization for Standardization, Geneva, CH, October 1986.

[33] ISO Central Secretary. Information technology — SGML support facilities — Registration procedures for public text owner identifiers. Standard ISO/IEC 9070:1991, International Organization for Standardization, Geneva, CH, April 1991.

[34] Norman Walsh, John Cowan, and Dr. Paul Grosso. A URN Namespace for Public Identifiers. RFC 3151, August 2001.

[35] Paul J. Leach, Rich Salz, and Michael H. Mealling. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, July 2005.

[36] Ronald L. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.

[37] Donald E. Eastlake 3rd and Paul Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174, September 2001.

[38] Michael H. Mealling. A URN Namespace of Object Identifiers. RFC 3001, November 2000.

[39] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[40] Henry Thompson, Paul V. Biron, David Peterson, Michael Sperberg-McQueen, Ashok Malhotra, and Sandy Gao. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. W3C recommendation, W3C, April 2012. https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/.

[41] Gregg Kellogg, Dave Longley, and Pierre-Antoine Champin. JSON-LD 1.1. W3C recommendation, W3C, July 2020. https://www.w3.org/TR/2020/REC-json-ld11-20200716/.

[42] Dan Brickley and Ramanathan Guha. RDF Schema 1.1. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[43] Luigi Asprino, Enrico Daga, Aldo Gangemi, and Paul Mulholland. Knowledge Graph Construction with a Façade: A Unified Method to Access Heterogeneous Data Sources on the Web. *ACM Trans. Internet Technol.*, 2022.

[44] Enrico Daga, Luigi Asprino, Paul Mulholland, and Aldo Gangemi. Facade-X: an opinionated approach to SPARQL anything. *CoRR*, abs/2106.02361, 2021.

[45] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, May 2011.

[46] Steve Faulkner, Erika Doyle Navara, Silvia Pfeiffer, Ian Hickson, Robin Berjon, Travis Leithead, and Theresa O'Connor. HTML5. W3C recommendation, W3C, March 2018. https://www.w3.org/TR/2018/SPSD-html5-20180327/.

[47] Richard Cyganiak and Leo Sauermann. Cool URIs for the Semantic Web. W3C note, W3C, December 2008. https://www.w3.org/TR/2008/NOTE-cooluris-20081203/.

[48] Paul C. Bryan, Kris Zyp, and Mark Nottingham. JavaScript Object Notation (JSON) Pointer. RFC 6901, April 2013.

[49] Norman Walsh, Paul Grosso, Jonathan Marsh, and Eve Maler. XPointer Framework. W3C recommendation, W3C, March 2003. https://www.w3.org/TR/2003/REC-xptr-framework-20030325/.

[50] Carlos A. Velasco, Philip Ackermann, and Johannes Koch. Representing Content in RDF 1.0. W3C note, W3C, February 2017. https://www.w3.org/TR/2017/NOTE-Content-in-RDF10-20170202/.

[51] Chris Bizer. Semantic Web Publishing Vocabulary (SWP) User Manual. Technical report, November 2006. http://wbsg.informatik.uni-mannheim.de/bizer/wiqa/swp/SWP-UserManual.pdf.

[52] Richard Tobin. An RDF Schema for the XML Information Set. W3C note, W3C, April 2001. https://www.w3.org/TR/2001/NOTE-xml-infoset-rdfs-20010406.

[53] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, October 2006.

[54] Juan Benet and Manu Sporny. The Multihash Data Format. Internet-Draft draft-multiformats-multihash-04, Internet Engineering Task Force, February 2022. Work in Progress.

[55] Joseph Reagle, Kelvin Yiu, David Solo, Frederick Hirsch, Donald Eastlake, Magnus Nyström, and Thomas Roessler. XML Signature Syntax and Processing Version 1.1. W3C recommendation, W3C, April 2013. https://www.w3.org/TR/2013/REC-xmldsig-core1-20130411/.

List of Figures

List of Listings

List of Tables

A. Attachments

A.1 Application source code

The source code of the application, as described in section 6.2, is located in the `/src` directory in the attached ZIP file. It also corresponds to the tag `v1.0` in the GitHub repository at `https://github.com/cermakmarek/SFI`. Additionally, the `/docs` directory stores the auto-generated structured documentation, replicated at `https://cermakmarek.github.io/sfi/docs/`. The file `/example.zip.ttl` is a copy of the code in section A.2.

A.2 Sample description of `Samples/zip/example.zip`

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix schema: <http://schema.org/>.
@prefix nfo: <http://www.semanticdesktop.org/ontologies
   ↪/2007/03/22/nfo#>.
@prefix nie: <http://www.semanticdesktop.org/ontologies
   ↪/2007/01/19/nie#>.
@prefix skos: <http://www.w3.org/2004/02/skos/core#>.
@prefix cnt: <http://www.w3.org/2011/content#>.
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix dt: <http://dbpedia.org/datatype/>.
@prefix sec: <https://w3id.org/security#>.
@prefix enc: <http://www.w3.org/2001/04/xmlenc#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

<data:application/octet-stream,HEAD%00%00%00%01>
  dcterms:extent "8"^^dt:byte;
  dcterms:hasFormat <data:application/x.sig.head,HEAD
     ↪%00%00%00%01>;
  a cnt:ContentAsBase64;
  skos:prefLabel "binary data (8 B)"@en;
  cnt:bytes "SEVBRAAAAAE="^^xsd:base64Binary.

<data:application/x.sig.head,HEAD%00%00%00%01>
  schema:encodingFormat <https://w3id.org/uri4uri/mime/
     ↪application/x.sig.head>;
  a schema:MediaObject;
  skos:prefLabel "HEAD object (8 B)"@en.

<file:///dir/example.png> at:pathObject <file:///example.png>.
<file:///example.png> at:extensionObject <https://w3id.org/
   ↪uri4uri/suffix/png>.
```

```
<file:///unk-binary.dat> at:extensionObject <https://w3id.org/
   ↪uri4uri/suffix/dat>.
<file:///unk-textual.txt> at:extensionObject <https://w3id.org/
   ↪uri4uri/suffix/txt>.

<ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=
   ↪application/octet-stream>
  at:digest <urn:sha256:276
     ↪UFBWZH3B8GHMKQ6EHC45Y2HLUL959MSW79NJZEZP23EPMFGQQ>;
  dcterms:extent "3441"^^dt:byte;
  dcterms:hasFormat <ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--
     ↪u6kNttFkj8cdDXLQI?ct=application/zip>;
  a cnt:ContentAsBase64;
  skos:prefLabel "binary data (3.36 KiB)"@en.

<ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=
   ↪application/zip>
  at:pathObject <file:///>;
  schema:encodingFormat <https://w3id.org/uri4uri/mime/
     ↪application/zip>;
  nie:interpretedAs <ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--
     ↪u6kNttFkj8cdDXLQI?ct=application/zip#/>;
  a schema:MediaObject,
    nfo:Archive;
  skos:prefLabel "ZIP object (3.36 KiB)"@en.

<ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=
   ↪application/zip#/>
  at:pathObject <file:///./>;
  a nfo:Folder;
  skos:prefLabel "/".

<ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=
   ↪application/zip#/dir>
  at:pathObject <file:///dir>;
  nie:interpretedAs <ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--
     ↪u6kNttFkj8cdDXLQI?ct=application/zip#/dir/>;
  nfo:belongsToContainer <ni:///sha-256;407
     ↪cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=application/
     ↪zip#/>;
  nfo:fileLastModified "2023-03-19T01:21:34.000000+01:00"^^xsd:
     ↪dateTime;
  nfo:fileName "dir";
  a nfo:FileDataObject,
    nfo:ArchiveItem;
  skos:prefLabel "/dir".
```

```
<ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=
   ↪application/zip#/dir/>
  at:pathObject <file:///dir/>;
  a nfo:Folder;
  skos:prefLabel "/dir/".

<ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=
   ↪application/zip#/dir/example.png>
  at:pathObject <file:///dir/example.png>;
  nie:interpretedAs <ni:///sha-256;FrAhAufBERWjweVgm65W-
     ↪LxtbrApgClbV-SbFS36IiY?ct=application/octet-stream>;
  nfo:belongsToContainer <ni:///sha-256;407
     ↪cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=application/
     ↪zip#/dir/>;
  nfo:fileLastModified "2023-03-19T01:21:34.000000+01:00"^^xsd:
     ↪dateTime;
  nfo:fileName "example.png";
  nfo:fileSize 819 ;
  a nfo:FileDataObject,
    nfo:ArchiveItem;
  skos:prefLabel "/dir/example.png".

<ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=
   ↪application/zip#/unk-binary.dat>
  at:pathObject <file:///unk-binary.dat>;
  nie:interpretedAs <data:application/octet-stream,HEAD
     ↪%00%00%00%01>;
  nfo:belongsToContainer <ni:///sha-256;407
     ↪cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=application/
     ↪zip#/>;
  nfo:fileLastModified "2023-03-19T01:20:34.000000+01:00"^^xsd:
     ↪dateTime;
  nfo:fileName "unk-binary.dat";
  nfo:fileSize 8 ;
  a nfo:FileDataObject,
    nfo:ArchiveItem;
  skos:prefLabel "/unk-binary.dat".

<ni:///sha-256;407cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=
   ↪application/zip#/unk-textual.txt>
  at:pathObject <file:///unk-textual.txt>;
  nie:interpretedAs <ni:///sha-256;YUt039MujchKe-
     ↪KG_6_Il2kbICwlZevbturkTmKvcoE?ct=text/plain>;
  nfo:belongsToContainer <ni:///sha-256;407
     ↪cJBH1yB2r5mCiqtajo3rs--u6kNttFkj8cdDXLQI?ct=application/
     ↪zip#/>;
  nfo:fileLastModified "2023-03-19T01:48:38.000000+01:00"^^xsd:
     ↪dateTime;
  nfo:fileName "unk-textual.txt";
  nfo:fileSize 7481 ;
  a nfo:FileDataObject,
    nfo:ArchiveItem;
  skos:prefLabel "/unk-textual.txt".

<ni:///sha-256;FrAhAufBERWjweVgm65W-LxtbrApgClbV-SbFS36IiY?ct=
   ↪application/octet-stream>
  at:digest <urn:sha256:
     ↪NQR2ZAKKAVWCGGXUS3UND7MCU358X7ANDAQ6BF9CWPX4GNEPZGDQ>;
  dcterms:extent "819"^^dt:byte;
  dcterms:hasFormat <ni:///sha-256;FrAhAufBERWjweVgm65W-
     ↪LxtbrApgClbV-SbFS36IiY?ct=image/png>;
  a cnt:ContentAsBase64;
  skos:prefLabel "binary data (819 B)"@en.

<ni:///sha-256;FrAhAufBERWjweVgm65W-LxtbrApgClbV-SbFS36IiY?ct=
   ↪image/png>
  schema:encodingFormat <https://w3id.org/uri4uri/mime/image/
     ↪png>;
  nfo:colorDepth 24 ;
  nfo:height 44 ;
  nfo:horizontalResolution 95.9866;
  nfo:verticalResolution 95.9866;
  nfo:width 144 ;
  a schema:MediaObject,
    schema:ImageObject,
    nfo:Image;
  skos:prefLabel
    "PNG object (144x44)"@en,
    "PNG object (819 B)"@en,
    "PNG object (144x44, 8-bit)"@en.

<ni:///sha-256;YUt039MujchKe-KG_6_Il2kbICwlZevbturkTmKvcoE?ct=
   ↪text/plain>
  at:digest <urn:sha256:
     ↪A8BE755MVD2GRSF5Z7APP7ZJ7RMNATTSSDJNLF8NC3J9WG9HB4QA>;
  dcterms:extent "7481"^^dt:byte;
  nie:hasPart <ni:///sha-256;YUt039MujchKe-
     ↪KG_6_Il2kbICwlZevbturkTmKvcoE?ct=text/plain#line=,1>;
  a cnt:ContentAsText;
  skos:prefLabel "text (7.31 KiB)"@en;
  cnt:characterEncoding "us-ascii".

<ni:///sha-256;YUt039MujchKe-KG_6_Il2kbICwlZevbturkTmKvcoE?ct=
   ↪text/plain#line=,1> rdf:value "[Header]".
```

```
<urn:sha256:276
   ↪UFBWZH3B8GHMKQ6EHC45Y2HLUL959MSW79NJZEZP23EPMFGQQ>
  a sec:Digest;
  sec:digestAlgorithm enc:sha256;
  sec:digestValue "407cJBH1yB2r5mCiqtajo3rs++u6kNttFkj8cdDXLQI
     ↪="^^xsd:base64Binary.

<urn:sha256:
   ↪A8BE755MVD2GRSF5Z7APP7ZJ7RMNATTSSDJNLF8NC3J9WG9HB4QA>
  a sec:Digest;
  sec:digestAlgorithm enc:sha256;
  sec:digestValue "YUt039MujchKe+KG/6/Il2kbICwlZevbturkTmKvcoE
     ↪="^^xsd:base64Binary.

<urn:sha256:
   ↪NQR2ZAKKAVWCGGXUS3UND7MCU358X7ANDAQ6BF9CWPX4GNEPZGDQ>
  a sec:Digest;
  sec:digestAlgorithm enc:sha256;
  sec:digestValue "FrAhAufBERWjweVgm65W+LxtbrApgClbV+SbFS36IiY
     ↪="^^xsd:base64Binary.
```