

# Building the BBC Music Website



Ashley Harris

C1413555

Module: CM3203

Supervisor: Alia I Abdelmoty

Moderator: Jing Wu

## Abstract

Linked open data has made it easier for developers to build domain-specific web applications which offer enhanced scalability over traditional Web 2.0 applications. By 'mashing up' data from multiple linked sources, these applications can offer large amounts of useful information to their users. The BBC Music site is a good example which uses data from MusicBrainz and DBpedia, adding greater value to their own content.

My goal for this project was to develop an application which can automatically query the web for data that can be used to build a website such as BBC Music. The results of these queries would then be presented on automatically generated web pages. To achieve this, I investigated the state of music-related Linked Data and methods that I could use to build such an application.

## Acknowledgements

I would like to thank my supervisor Dr. Alia Abdelmoty for her guidance, feedback and reassurance throughout the entirety of the project. I would also like to thank my family and friends for all their encouragement and support.

## Table of Contents

Abstract.....	2
Acknowledgements.....	2
Table of Figures.....	5
1. Introduction .....	6
1.1 Project Overview .....	6
1.2 Aims and Objectives .....	7
1.3 Intended Audience and Beneficiaries .....	7
1.4 Restrictions and Assumptions .....	7
2 Background .....	8
2.1 The Semantic Web .....	8
2.2 Linked Data.....	8
2.2.1 Linked Open Data Cloud .....	9
2.2.2 Uses of the Linked Data Cloud .....	9
2.2.3 Creating Linked Data.....	9
2.3 Ontologies .....	10
2.3.1 Retrieving Data .....	12
2.4 Data Sources.....	13
2.4.1 MusicBrainz.....	13
2.4.2 DBpedia .....	14
2.4.3 Music Ontology .....	16
2.4.4 BBC Ontologies and the BBC Music Site .....	18
2.4.5 Using Data Sources to Build Application.....	18
2.5 Tools Used to Build Application .....	19
2.6 Assessing Usability of Current Data Sources.....	20
2.6.1 Querying DBpedia .....	20
2.6.2 Querying MusicBrainz .....	23
3 Specification and Design .....	25
3.1 Requirements .....	25
3.1.1 Functional Requirements.....	25
3.1.2 Non-functional requirements .....	25
3.2 Entity Relationship Diagram.....	26
3.3 Data Collection System .....	27
3.4 User interface .....	29
3.4.1 Wireframes .....	29

3.4.3	Choice of External Links for Application .....	32
4	Implementation .....	34
4.1	Choice of platform.....	34
4.3	Home Page .....	34
4.4	Artist Pages.....	37
4.5	Album Pages .....	39
4.6	Notable sections.....	40
4.6.1	Retrieving Band Members .....	40
4.6.2	Handling Genres.....	42
4.6.3	Creating RDF .....	43
4.6.4	Getting a track list.....	45
4.6.5	Dealing with ambiguity when getting descriptions .....	47
4.6.6	Key Errors .....	47
4.6.7	Refactoring.....	48
4.6.8	Implementation on a Web Server.....	49
5	System Testing and Evaluation .....	50
5.1	Testing methodology.....	50
5.2	Test Cases .....	50
5.3	Meeting Requirements .....	53
5.3.1	Functional Requirements.....	53
5.3.2	Non-functional requirements .....	54
5.4	Accessibility and Usability .....	54
5.5	Performance and Error Handling .....	55
5.6	Viability of Implementation Method .....	56
6	Future Work.....	57
6.1	Obtaining Data from More Sources .....	57
6.2	Icons Indicating Data Provenance .....	58
6.3	Adding a Database to the Application .....	58
6.4	Wider Search Capability .....	59
6.5	More links and DBpedia content.....	60
6.6	Identification of missing content .....	60
7	Conclusions .....	62
7.1	Reviewing Aims and Objectives .....	62
7.1.1	Research.....	62
7.1.2	Identifying Sources.....	62

7.1.3	Designing an Application .....	63
7.1.4	Testing Application .....	63
7.2	Achievements .....	63
8	Reflection on learning .....	65
9	Bibliography .....	67

## Table of Figures

Figure 1 - Example of a taxonomy .....	10
Figure 2 - Snippet from the Entity Relationship Diagram representing the schema of the MusicBrainz database .....	13
Figure 3 - Example of a Wikipedia infobox .....	15
Figure 4 - DBpedia home page .....	20
Figure 5 - DBpedia search page .....	21
Figure 6 - Example of a DBpedia resource represented as an HTML page on the site .....	21
Figure 7 - Interface for SPARQL endpoint hosted by DBpedia .....	22
Figure 8 - An example SPARQL query .....	22
Figure 9 - A MusicBrainz artist information page .....	23
Figure 10 - A MusicBrainz release-list page (Top) and an individual release information page (Bottom) .....	24
Figure 11 - Entity Relationship Diagram (ERD) for my application .....	26
Figure 12 - Flow diagram to illustrate the procedures taken by the application's data collection system .....	28
Figure 13 - Home page of the application .....	29
Figure 14 - Artist information page from the application .....	30
Figure 15 - Album information page from the application .....	31
Figure 16 - Genre information page from the application .....	32
Figure 17 - Screenshot of the application's home page with no search results .....	35
Figure 18 - Screenshot of the application's home page once a search query has been submitted .....	35
Figure 19 - Screenshot of an artist information page .....	37
Figure 20 - Screenshot of an album information page .....	39
Figure 21 - Code from get_dbpedia_genres function in views.py .....	41
Figure 22 - Code from the artist_info function in views.py .....	42
Figure 23 - Examples of standard RDF/XML (Top) and Turtle serialization (Bottom) .....	44
Figure 24 - An example of a Turtle file produced by the application .....	44
Figure 25 - RDFa embedded in a BBC Music artist web page .....	45
Figure 26 - RDFa embedded within an Artist information page on the application I developed .....	45
Figure 27 - Code from the get_location method in views.py .....	48
Figure 28 - Artist information page with extended release information .....	55
Figure 29 - Illustration of how data provenance icons would be displayed .....	58
Figure 30 - Open Knowledge Graph Properties embedded in a BBC Music artist information page .....	65

# 1. Introduction

## 1.1 Project Overview

The BBC Music website aims to provide a guide to music content from the BBC, including profiles of artists who appear on BBC programmes [1]. To provide additional information related to the artists featured in their own content (e.g. radio interviews, television performances), the BBC Music site utilises Linked Data to great effect.

The purpose of this project was to investigate how Linked Data could be used to build an application which allows users to search for music artists in the same way that they can on the BBC Music site but offer more biographical and discography related information. The BBC does not offer much of this information due to their focus on their own television and radio content.

After researching the current state of openly available Linked Data sets regarding music, the goal was to create a web application which allowed users to easily retrieve information about their favourite artists. The application also needed to adhere to good practices regarding Linked Data, which means relationships between entities on the site would be clearly defined. In addition to displaying the data itself from linked sources, including as many hyperlinks to additional sources of information as possible was important to provide a larger scope of information than the BBC Music site offers.

The application I have developed aggregates data from three different sources: DBpedia, MusicBrainz and BBC Music itself. In this report I will explain how these Linked Data sets are formed, how I have used them to generate content for my application and any advantages or disadvantages that the data sources presented during development.

Building a website such as BBC Music requires databases and servers with large storage capacity and high access speeds due to the sheer volume of information and users. By utilising Linked Data sets it was possible for me to build an application that does not require its own databases. Instead, queries for data are executed dynamically as the user 'clicks' through the site and data is retrieved from external data servers. I could access such data in this way due to SPARQL endpoints hosted by sources or web APIs. At the end of this report I will summarise whether this is an effective way of developing an application related to music, given the state of music related Linked Data.

For instance, the quantity and quality of information available varies between data sets and subject matter. Using Linked Data to populate the website therefore presented challenges. There were issues regarding data integrity, consistency and provenance, mainly because such data sources are contributed to by the public. This report will discuss the quality and availability of music related Linked Data assessing its suitability for building an application such as the one I have developed for this project.

## 1.2 Aims and Objectives

During the initial planning phase, the following aims were identified as the most important for the project:

- Study different approaches available for building a linked data driven web application and choose a suitable platform for implementation.
- Identify sources of data and build a data collection system to query multiple data sources, integrate and pre-process the results for publishing.
- Design and implement an interface component of the application which allows the filtering and extraction of specific data sets and the presentation of the data in an appropriate format.
- Test and evaluate the system with multiple data sets on the linked data cloud. Produce web pages that are automatically generated based on content the user is looking for.

In the conclusion of this report I will discuss and evaluate my progress towards achieving these four aims. (Section 7.1 Reviewing Aims and Objectives)

## 1.3 Intended Audience and Beneficiaries

This report will be useful to anyone who is interested in utilising Linked Data related to music because I investigated the quality, accessibility and usefulness of various sources. Furthermore, this report will be useful for anyone looking for guidance on developing a web application which utilises Linked Data and various APIs to deliver a service. I will describe how my application is put together, highlighting any significant challenges I faced during implementation and what I did to overcome such problems.

## 1.4 Restrictions and Assumptions

- Due to time constraints, the web application will be developed only for desktop machines and not mobile devices.
- The user of the application must be connected to the Internet, no information will be available on the application offline.
- The application developed is only a prototype and therefore, only versions running on a local Django server with a Chrome Web Server extension installed offer full functionality. A public version has been implemented on a web server hosted at [www.pythonanywhere.com](http://www.pythonanywhere.com) but one feature is currently absent from this version (see Section 4.6.8 Implementation on a Web Server for more information).
- For the application to work correctly, the data sources used to populate the application must be online and the APIs or SPARQL endpoints required to access them must be functioning correctly.
- For the application to provide up to date information, it is assumed that the data sources used to populate the application are regularly updated.

## 2 Background

### 2.1 The Semantic Web

The World Wide Web was first developed at CERN in 1990 by Tim Berners-Lee and Robert Cailliau. Even early proposals by Berners-Lee from this time hint at ambitions beyond the simple document formatting language known as HTML. In one such proposal, Berners-Lee talks of problems such as scattered information and incompatible document types as well as depicting the web as connections between concepts, people and organisations, instead of just documents alone. Such connections between multiple entities with their meaning explicitly defined is a core concept of the Semantic Web, a phrase coined by Berners-Lee coined during a keynote speech in 1995. [2]

The Semantic Web is a web of data described and linked in ways establish context or meaning. There are several formal standards that make up the Semantic Web, which make it readable for machines and humans alike. This is achieved by explicitly defining the relationships between data items which enables information to be exchanged between applications. This is known as interoperability. [3]

The idea of turning data relationships from implicit to explicit is what makes the Semantic Web and ontologies so useful, especially in fields such as Artificial Intelligence, as machines cannot grasp context and meaning like humans can. Ontologies are used to formally define concepts and properties within a domain (to be explained in Section 2.3 Ontologies).

### 2.2 Linked Data

Semantic Web concepts are realised through Linked Data. It useful to provide links between data so that a web user can find data about one entity and then follow links related to this entity to obtain more data. Linked Data therefore provides opportunities to enrich and extend the information offered by library data, especially bibliographic data [4]. This concept can also be applied to web applications. By adding extra information to databases which complements original data, the scope of an application's dataset can be expanded, providing a richer variety of knowledge as a result.

In 2010, Berners-Lee created a five-star rating system for Linked Open Data to encourage people to make data more 'powerful' and 'easier to use':






	Available on the web (whatever format) but with an open licence, to be Open Data
	Available as machine-readable structured data (e.g. Excel instead of image scan of a table)
	as (2) plus non-proprietary format (e.g. CSV instead of Excel)
	All the above plus: Use open standards from W3C (RDF and SPARQL) to identify things, so that people can point at your stuff
	All the above, plus: Link your data to other people's data to provide context

Table 1 - Five Star Rating System for Linked Open Data



### 2.2.1 Linked Open Data Cloud

The Linked Open Data cloud (LOD) is a collection of interrelated datasets on the Web which are free to access for anyone [5]. It is the sort of solution Tim Berners-Lee has been encouraging people to move towards with his five-star rating system for publishing open data. As of August 2014, the LOD cloud has 1,014 datasets covering a variety of topics including geographic, governmental and social media data. [6]

### 2.2.2 Uses of the Linked Data Cloud

The rich variety of data available from the LOD cloud makes it useful for building applications which cover a wide range of topics as the data can all be accessed the same way. Usually structured data on websites is only accessible via an API, so without Linked Data, applications built to obtain data from multiple sources would have to rely on the use of several APIs. As APIs can vary greatly in implementation this can make their utilisation difficult. For example, several different programming languages, which may be incompatible, may be required within the program to implement different APIs. [7]

The BBC Music website is a great example of how Linked Data has been used to enhance the levels of information provided by the service. The purpose of the site is to host all BBC content related to music, such as snippets of radio and TV broadcasts including interviews or performances from the world of music. The site uses metadata and unique identifiers for music artists from musicbrainz.org whereas artist biographies come from DBpedia.org. Both sites host data sets in the LOD cloud, so BBC Music is using Linked Data to complement their own content.

### 2.2.3 Creating Linked Data

In 2006 Berners-Lee proposed four rules which can be used to effectively turn ordinary data into Linked Data:

1. Use URIs as names for things
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF\*, SPARQL)
4. Include links to other URIs, so that they can discover more things. [8]

One of the most important Semantic Web standards mentioned in these rules is the Resource Description Framework (RDF). Resources on the web are any object or concept that can be 'named'. In RDF they are represented by Uniform Resource Identifiers (URI) which are mentioned in each of the four rules. All URIs are unique and use the following format:

scheme ':' ['/' authority '/'] [path] ['?' query] ['#' fragment]

The idea of the framework is that 'connections between resources are represented by predicates or properties'. Properties are also URIs, where the authority and path define a schema or ontology. [3]

Web pages that use RDF to add semantics to the data that they represent can be interpreted by machines. The SPARQL Protocol and RDF Query Language (SPARQL) has been designed to 'query data conforming to the RDF data model'. This was important for this

project as websites hosting SPARQL endpoints provided me with a way to query Linked Data sets to retrieve content for populating the application.

### 2.3 Ontologies

Ontologies are used to explicitly define concepts. They contain classes which describe the contents of a domain. If necessary, such classes can be divided into sub classes which are more specific. For example, an ontology defining vehicles could have the following classes and subclasses:

- Vehicle
  - Land Vehicle
    - Motorcycle
    - Car
    - Van
    - HGV
  - Air Vehicle
    - Aeroplane
    - Helicopter
  - Water Vehicle
    - Boat
    - Submarine

This taxonomic hierarchy could be extended much further to include many more classes of vehicle. To make it a usable ontology, relationships and properties related to each class should be defined:

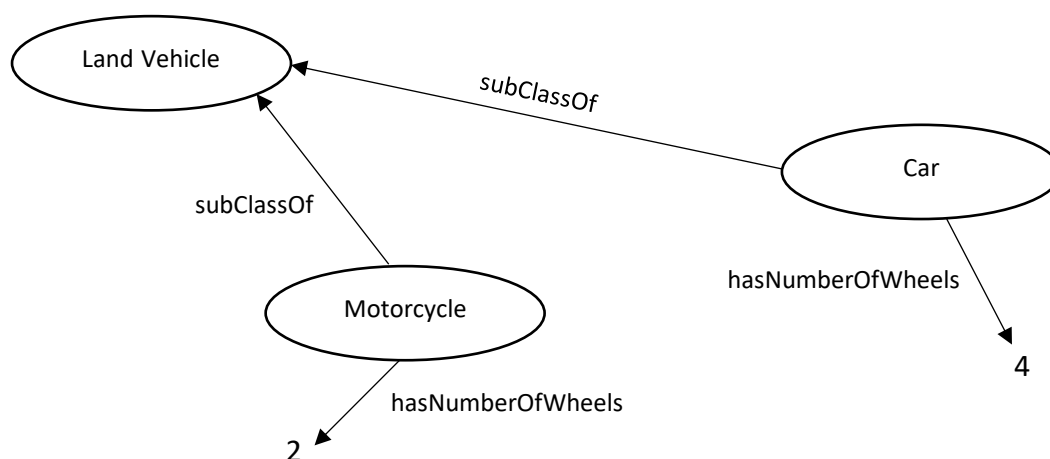


Figure 1 - Example of a taxonomy

Figure 1 shows how the classes Car and Motorcycle are subclasses of the Land Vehicle class and that they both have a property defining what 'number of wheels' an entity must have to be a member of the class. This graph is just a small example of what could be included in an ontology such as this, these three classes could have many more properties and relationships associated with one another.

This example demonstrates how important it is to clearly define the scope of an ontology before it is created. Ontologies can quickly grow and become confusing if they describe

entities and relationships of too many different types. There are also many ways of defining concepts and relationships to build ontologies. For these reasons, ontologies are best developed to only cover very specific fields. If required, an ontology can then be used in conjunction with others to meet the demands of a specific application. The Linked Data Cloud utilises this ability for ontologies to be combined.

As there are over 1,000 datasets within the Linked Data cloud which cover a wide range of topics, they all require their own ontologies and ways of mapping data entities to one another. However, as these sources are interlinked, the use of these ontologies is often shared across datasets. The ontologies make this interlinking possible.

**Three most used predicates for interlinking by category.**

Category	Predicate	Usage	Category	Predicate	Usage
social web	foaf:knows	60.27%	life sciences	owl:sameAs	52.17%
	foaf:based_near	35.69%		rdfs:seeAlso	48.48%
	sioc:follows	34.34%		dct:creator	21.74%
publications	owl:sameAs	32.20%	government	dct:publisher	47.57%
	dct:language	25.42%		dct:spatial	30.10%
	rdfs:seeAlso	23.73%		owl:sameAs	24.27%
user-generated content	owl:sameAs	53.13%	geographic	owl:sameAs	64.29%
	rdfs:seeAlso	21.88%		skos:exactMatch	21.43%
	dct:source	18.75%		skos:closeMatch	21.43%
media	owl:sameAs	81.25%	cross-domain	owl:sameAs	80.00%
	rdfs:seeAlso	18.75%		rdfs:seeAlso	52.00%
	foaf:based_near	18.75%		dct:creator	20.00%

Table 2 - Most used predicates for interlinking on DBpedia

[6]

Table 2 is from the State of the LOD Cloud document from 2014. It shows the three most-used ontology predicates for each category of data set on the cloud. It shows that more than one type of ontology is used in every instance. There are examples of the Friend Of A Friend (FOAF), Web Ontology Language (OWL) and Resource Description Framework Schema (RDFS) schema ontologies amongst others. Use of these predicates from different vocabularies allows a variety of RDF triples to be created, thus linking items from multiple datasets together.

### 2.3.1 Retrieving Data

As mentioned in Section 2.2.3 Creating Linked Data, Linked Data can be retrieved using SPARQL queries. The structure of SPARQL queries is much like that of SQL queries except they are used for retrieving data stored in an RDF format instead of traditional relational databases.

There are four different query variations for different purposes in SPARQL:

- **SELECT:** used to extract raw values from a SPARQL endpoint. Results are returned in a table format.
- **CONSTRUCT:** used to extract information from a SPARQL endpoint and turn it into valid RDF.
- **ASK:** used to provide a simple True/False result for a query to a SPARQL endpoint.
- **DESCRIBE:** used to extract an RDF graph from a SPARQL endpoint

Here is an example of a SPARQL query:

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

SELECT ?name
       ?email
WHERE
{
    ?person a          foaf:Person .
    ?person foaf:name  ?name .
    ?person foaf:mbox  ?email .
}
```

Which results in something like this:

name	email
"Ashley Harris"	<mailto:aharris@example.com>
"Thom Yorke"	<mailto:yorket@example.org>

[9]

The three lines in the **WHERE** clause of this query are RDF triples. The purpose of the **PREFIX** statement is to simplify the query and make it more human readable. So, the predicate **foaf:name** is the same as <http://xmlns.com/foaf/0.1/name>. Variables in SPARQL queries are indicated by question marks at the start of the variable.

To use SPARQL to extract RDF data from knowledge graphs, the user must use a SPARQL endpoint. The W3C provides a list of currently active endpoints allowing access to datasets in the Linked Open Data cloud.

Writing SPARQL queries like this is not straightforward for many web users with limited to no knowledge of this subject area, so one of the goals for the project was to develop an application with an easy to use interface which hides the SPARQL queries from the user.

## 2.4 Data Sources

### 2.4.1 MusicBrainz

Before I began development of my application I carried out research into available datasets related to Music. I began by investigating the source of metadata used by the BBC Music site, MusicBrainz, a site which is contributed to by its community of users. Biographies of artists on BBC Music come from MusicBrainz via Wikipedia. [1]

After investigation of the MusicBrainz site I decided I wanted to use it as the primary source of data for my application due to the volume of data it holds. As of 23<sup>rd</sup> March 2018, the site features 1,331,212 artists and 1,571,350 *release groups* [10]. *Release groups* on MusicBrainz can include singles, EPs, studio albums, live albums etc. The *release groups* themselves then contain '*releases*' which can be thought of as 'real-world' objects which can be bought in stores [11]. For example, a *release group* for a particular album may contain several different *releases* as the album may have been marketed in different countries via different labels, leading to various release dates and mediums (CD, vinyl, digital release). Figure 2 illustrates the properties of each of these entities and the relationships between them.

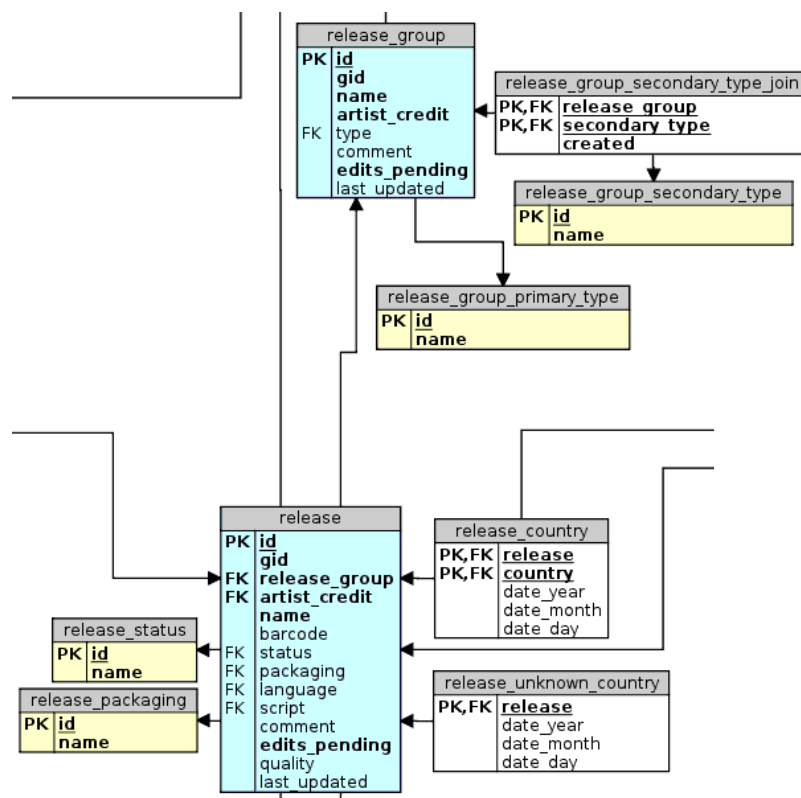


Figure 2 - Snippet from the Entity Relationship Diagram representing the schema of the MusicBrainz database

[11]

I decided to look at whether the MusicBrainz site offered a SPARQL endpoint to an RDF representation of their data so that my application could retrieve it via SPARQL. MusicBrainz provides unique identifiers for music artists, albums and tracks and has therefore been widely used as a source for music-related URIs in the Linked Data community. Work has been done in the past to publish the MusicBrainz database as Linked Data in a project called

LinkedBrainz. As people were creating new URIs based on MBIDs, the goal of the project was to reduce the need for duplication by providing Linked Data directly from MusicBrainz. [12]

Information on the MusicBrainz site indicates there are four available RDF mappings that were produced because of this project. However, links to two of these sources ultimately lead to web pages that no longer exist. Furthermore, a link to data dumps based on the original RDF service used by MusicBrainz which are said to be ‘infrequently updated’, result in an error (404 URL not found on this server) [13]. These resources are most likely unavailable because the LinkedBrainz project ended nearly seven years ago, in July 2011. However, one of these sources is still available, the D2R server or [www.dbtune.org/musicbrainz](http://www.dbtune.org/musicbrainz).

DBTune shows all information and relationships between entities on MusicBrainz as Subject, Property, Value triples using either URIs or string literals. It also has a functioning SPARQL endpoint, so I would be able to write my own SPARQL queries to fetch the information I required to populate the application. However, there remained the issue of the LinkedBrainz project no longer being active. I searched DBTune for music artists that I knew came to prevalence in years after 2011 and found that they had no DBTune pages, and therefore no RDF triples that could be used. This was a major setback as my original plan was to implement the application so that it retrieved MusicBrainz data using SPARQL queries. I wanted the site to display up to date information, so I did not want to settle for using the out of date RDF on DBTune.

#### 2.4.2 DBpedia

With no way of getting music related RDF from MusicBrainz I moved on to investigating DBpedia. Launched in 2007, DBpedia sits at the centre of the LOD cloud. It provides structured information resembling an open knowledge graph (OKG) which is extracted from various Wikimedia projects. It describes over 4.5 million entities, 411,000 of which are creative works including 123,000 music albums [14]. Apple, Google and IBM are just a few of the notable businesses that have benefitted from DBpedia’s contributions, especially in the artificial intelligence field. It is also a ‘staple for academic pursuits in areas such as Ontology Design, Machine Learning and Natural Language Processing’. [15]

DBpedia has its own ontology which has classes to represent data items from infoboxes on Wikipedia pages. These are tables on Wikipedia with a consistent format, which usually appear on the right-hand side of the page (see Figure 3).

Arctic Monkeys	
	
Arctic Monkeys performing at Roskilde Festival on 5 July 2014.	
Background information	
<b>Also known as</b>	Death Ramps
<b>Origin</b>	Sheffield, England
<b>Genres</b>	Indie rock • garage rock • post-punk revival
<b>Years active</b>	2002–present
<b>Labels</b>	Domino
<b>Associated acts</b>	Miles Kane • Richard Hawley • The Last Shadow Puppets • Josh Homme • Dead Sons • Dizzee Rascal • Bill Ryder-Jones • Queens of the Stone Age • Iggy Pop • Alexandra Savior • Mini Mansions
<b>Website</b>	<a href="http://arcticmonkeys.com">arcticmonkeys.com</a> 
<b>Members</b>	Alex Turner Matt Helders Jamie Cook Nick O'Malley
<b>Past members</b>	Andy Nicholson

Figure 3 - Example of a Wikipedia infobox

Due to variations between such infoboxes, DBpedia describe the ontology as being ‘based on the most commonly used infoboxes’. Being a cross domain ontology, it is the biggest ontology that was used to develop the application with around 700 classes described by 2,795 different properties. [16]

Unlike MusicBrainz, DBpedia provides information about many things other than music, so I wanted to use it to link interesting information to facts about music artists. For instance, I believed that I could use it to provide detailed descriptions about the town, city or country that a music artist originated from, instead of simply retrieving the name of such locations. Furthermore, in comparison to MusicBrainz RDF available at DBTune, the data on DBpedia is more up to date and still being updated. Originally, Wikipedia dumps were taken periodically every few months so DBpedia was always 6-18 months behind updates applied to Wikipedia content [17]. However, the increased use of DBpedia resulted in the need for the DBpedia-Live project to be set up. This system ensures that DBpedia is updated daily,



with thousands of entities being updated every 5 minutes. Update statistics can be viewed here: <http://dbpedia-live.openlinksw.com/live/>.

Despite the large variety DBpedia resources can offer, writing SPARQL queries to DBpedia in my early research made it apparent that DBpedia could not be used to provide the main discography content for the web application I was developing. This was due to the inconsistency between resources representing musical artists and albums. DBpedia can also be subject to ‘factual errors inherited from Wikipedia’ and ‘incomplete mappings from Wikipedia infoboxes to the DBpedia ontology’. [18]

For example, DBpedia ontology predicates are used in different ways on different artist resource pages. The ‘*dbo:artist*’ predicate is used on artist pages to state that they are the artist of several DBpedia resources. These resources are usually a random collection of singles, album tracks, studio albums, EPs and live albums. As there is no logic behind what data is applied to this property, it varies widely from artist to artist. If I was to use a SPARQL query to retrieve this data to make up the main discography data of my application, some very complicated and sophisticated code would be required to identify what type of musical release each resource is and sort the data accordingly. This task is made more complicated by the fact that DBpedia resources do not follow any naming conventions. In the final solution, this proved an issue when obtaining the correct DBpedia resources for artists before then moving on to collecting the artist’s data from their resource. Section 4.6.5

Dealing with ambiguity when getting descriptions details what I did during implementation in an attempt to overcome this problem.

Even if it was achievable within the given time-frame of the project to produce an algorithm that could sort through all the artist’s releases returned from DBpedia, there would still be many absent releases as they are not always mapped to this property. In addition to this, DBpedia resources for artists also use the property ‘*dbo:musicalArtist*’ which is another random list of musical releases of varying types.

As DBpedia would not provide the core discography information for the web application, I carried on searching for music related RDF sources online. The only results other than DBTune and the LinkedBrainz project was the Music Ontology (<http://musicontology.com/>).

#### 2.4.3 Music Ontology

Built on RDF, the Music Ontology is a model for publishing structured music-related data on the web. Developed once the use of the Internet started to radically change the music industry, the Music Ontology was an attempt to link all machine-readable musical information and express all the relations between such information. On their site, the curators of the Music Ontology state that they used RDF for the ontology as it is extensible, thus ‘allowing Music Ontology-based data to be mixed with claims made in any other RDF vocabulary’ [19]. As already discussed, extensibility is a crucial part of realising Linked Data. RDF also allows for RDFa to be used to embed music related data in web pages. An explanation of RDFa and its uses is found in Section 4.6.3 Creating RDF of this report.

Examples of Music Ontology classes for describing musical artists and works include MusicArtist, Record, ReleaseType, Instrument and Genre. They also have more technical



classes such as AudioFile, CD and SACD (Super Audio Compact Disc) as well as event-based classes such as Show and Festival.

In 2011, MusicBrainz moved to a new data model called Next Generation Schema (NGS). MBIDs, a key aspect of MusicBrainz used to uniquely identify every resource on their site, were introduced with the NGS. BBC Music uses these same IDs for corresponding artists which can be viewed on their site. Table 3 is from a MusicBrainz Wiki page. It shows how the LinkedBrainz project used the Music Ontology (abbreviated to 'mo' in the following table) to map core entities from the NGS to RDF. Use of these mappings is evident in resource pages on the DBTune site. These mappings were also useful for the implementation of an extra feature in my application which allowed RDF triples to be created for musical entities represented in my own application (See Section 4.6.3 Creating RDF).

Type	NGS	RDF
core entities with mbids	'artist'	' <a href="#">mo:MusicArtist</a> ' (type=1 => ' <a href="#">mo:SoloMusicArtist</a> ', type=2 => ' <a href="#">mo:MusicGroup</a> ')
	'release'	' <a href="#">mo:Release</a> ' and ' <a href="#">mo:ReleaseEvent</a> ' (currently 1:1 relations)
	'release_group'	' <a href="#">mo:SignalGroup</a> '
	'recording'	' <a href="#">mo:Signal</a> '
	'label'	' <a href="#">mo:Label</a> '
	'work'	' <a href="#">mo:MusicalWork</a> '
entities with a URI	'url'	'rdfs:seeAlso' or 'owl:sameAs' links to <a href="#">DBpedia</a> etc.
	'track'	' <a href="#">mo:Track</a> ' with a URI that includes the track.id
	'medium'	' <a href="#">mo:Record</a> ' with a URI that includes the medium.id and various ' <a href="#">mt</a> ' instances (related via the <a href="#">mo:media_type</a> property)
objects	'tracklist'	' <a href="#">mo:Record</a> ' (tracklist currently are not treated separately, parts of this table are mapped to property values of the related <a href="#">mo:Record</a> instances)
	'artist_credit'	collapse into <a href="#">foaf:maker</a> relations

Table 3 - Mappings of MusicBrainz entities to RDF from the LinkedBrainz project

[20]

Researching the NGS led me to a different way of retrieving MusicBrainz discography data than originally planned. MusicBrainz has a Python API called musicbrainzngs 0.6 (MBAPI). The documentation (<http://python-musicbrainzngs.readthedocs.io/en/v0.6/>) details methods for retrieving information, searching the site and contributing to it. Many of the functions available rely on MBIDs to be passed as parameters. I therefore decided that I would use this API to gather the main discography information for the web application being developed and that I would use corresponding MBIDs on my site so that I could use the MBAPI to its full potential. This proved more effective for gathering the main discography information of artists as the data was always complete and returned in a predictable form

(always a list of Python dictionaries). Developing a program to extract the information I required from these dictionaries was a much more viable solution than using SPARQL queries alone to DBpedia (As discussed in Section 2.4.2 DBpedia).

#### 2.4.4 BBC Ontologies and the BBC Music Site

The BBC have created their own set of ontologies; their specification can be found at: <https://www.bbc.co.uk/ontologies>. There are 14 ontologies in the set which are all designed to support BBC Sport, Education, Music and News projects. The organisation first started using ontologies to publish data during the 2010 FIFA World Cup. The framework they developed during this time facilitated the ‘publication of automated metadata-driven web pages’ which could ‘automatically aggregate and render links to relevant stories’. This meant that it saved employees time and provided users with a greater variety of information [21]. This website was able to handle on average 1 million SPARQL queries per day [22]. The architecture for this system was referred to as a Dynamic Semantic Publishing Framework and it facilitates content navigation, re-use and visibility. This initiative would not have been possible without the use of the BBC Ontologies which allowed content to be explicitly described with properties and linked with explicit relationships.

There are only three BBC datasets on the LOD cloud: BBC Music, BBC Programmes and BBC Wildlife Finder. From these three, only the BBC Music set is relevant to the work I was carrying out. I tried to see if there was a way to download data dumps of the BBC Music site but could not find anything of use on the BBC Music site itself or <https://old.datahub.io/dataset/bbc-music>. The LOD cloud website’s information page about BBC Music stated that access was through a SPARQL endpoint. The link provided with this information was no longer active but fortunately, I found a link to an active endpoint for BBC Programmes and Music at <https://www.w3.org/wiki/SparglEndpoints>. This meant I had a way of accessing BBC Music data but no way to clearly see the relationships and properties assigned to resources like you can on DBpedia (see Figure 6 for an example). Reviewing the HTML source code of BBC Music artist pages did not reveal any explicit use of BBC ontologies either, just RDFa (more on this in Section 4.6.3 Creating RDF).

From this research I can only presume that the BBC use their ontologies privately to organise content on their site or that they used to publish RDF relating to their websites which was linked using the ontology, but these data dumps are no longer available. As I could not see data properties explicitly as I could on DBpedia I looked for guides to retrieving BBC data via SPARQL. Fortunately, OpenLink Software, who provide the SPARQL endpoint I used for DBpedia and BBC Music, provide a tutorial page with examples of SPARQL queries for retrieving BBC data (tutorial found at: [https://virtuoso.openlinksw.com/tutorials/spargl/SPARQL\\_Tutorials\\_Part\\_3/SPARQL\\_Tutorials\\_Part\\_3.html](https://virtuoso.openlinksw.com/tutorials/spargl/SPARQL_Tutorials_Part_3/SPARQL_Tutorials_Part_3.html)). I ended up using one of these examples in my application to provide extra functionality in the form of links to album reviews (see Section 4.5 Album Pages).

#### 2.4.5 Using Data Sources to Build Application

From this initial research I concluded that the application could provide vast amounts of information about music artists without the user having to navigate their way through several pages (like they do on MusicBrainz) to find such information. By complementing the comprehensive discography data available on MusicBrainz with regularly updated DBpedia

facts and relationships, the scope of information available via the application would be larger than what it may be possible to offer if the application only had one data source.

I also wanted to utilise the BBC Ontologies in some way to retrieve data from the BBC Music site via SPARQL queries, which could complement the core MusicBrainz and DBpedia data on the application.

## 2.5 Tools Used to Build Application

The application was developed using Python 3.6.3 and Django, an open source web development framework for Python. I chose to use Python for several reasons. Firstly, it allowed me to utilise the MBAPI which was essential for retrieving core discography data for the web application. Secondly, I am very familiar with it as I have used it more than any other language in my time at Cardiff University. Another reason for using Python was the 'SPARQLWrapper' package which I downloaded. Utilising this package was necessary for retrieving data from the Linked Data sources (DBpedia and BBC Music) that I would be using for my application. Provided an active SPARQL endpoint was specified as a variable, the SPARQLWrapper package allowed queries to be embedded in the Python functions that formulate the back-end of the application.

Using Django was desirable because it allowed me to run a local web server on my own machine which gave me flexibility during development as well as the ability to quickly see how changes to my code had affected the application. For example, using a more traditional method of web development and accessing the School's web server using WinSCP would have required me to manually transfer files across to the server every time I made a change. This would have been more time-consuming than using Django, which automatically refreshes the server when changes to relevant files are made.

Furthermore, when I first attempted to write Python code to be used on the School's web server, error pages were returned with no indication of what part of the code was causing the web page not to render. On the other hand, Django displays comprehensive error information and the values of certain variables at certain points of code execution, which made debugging easier.

There are other languages and packages that can be used for Semantic Web Programming. Apache Jena for example, is an open-source Java framework allowing Semantic Web constructs to be translated into Java classes and objects. Resources and properties can be defined, and relationships inferred using a reasoner. This framework would therefore have been useful for my project if I was storing the data retrieved and there was a need to develop my own ontology. Furthermore, examining the Jena documentation, the process of connecting to and using a SPARQL endpoint looks more complicated and involved many more lines of code than the very simple intuitive method offered by Python's SPARQLWrapper package.

If I was required to implement my own RDF store, instead of retrieving web page data each time a user interacts with page, then I could have used the Sesame Framework. It is a MySQL based method of storing RDF, which requires two tables, one listing all the resources in

a dataset with unique IDs and another defining triples by specifying which elements from the resources table make up the subject, predicate and object. Each triple is then given its own ID. As data from my application was all retrieved in real-time there was no need to use Sesame. [3]

## 2.6 Assessing Usability of Current Data Sources

For the application I developed to be a gateway to music-related Linked Data which was simple to use, I needed to analyse how easy it is to use DBpedia and MusicBrainz. By identifying any strengths or weaknesses that they exhibit, I would be able to build a higher-quality application.

### 2.6.1 Querying DBpedia

Although it offers a vast amount of data, DBpedia is not very user friendly. Figure 4 shows the homepage for the site.

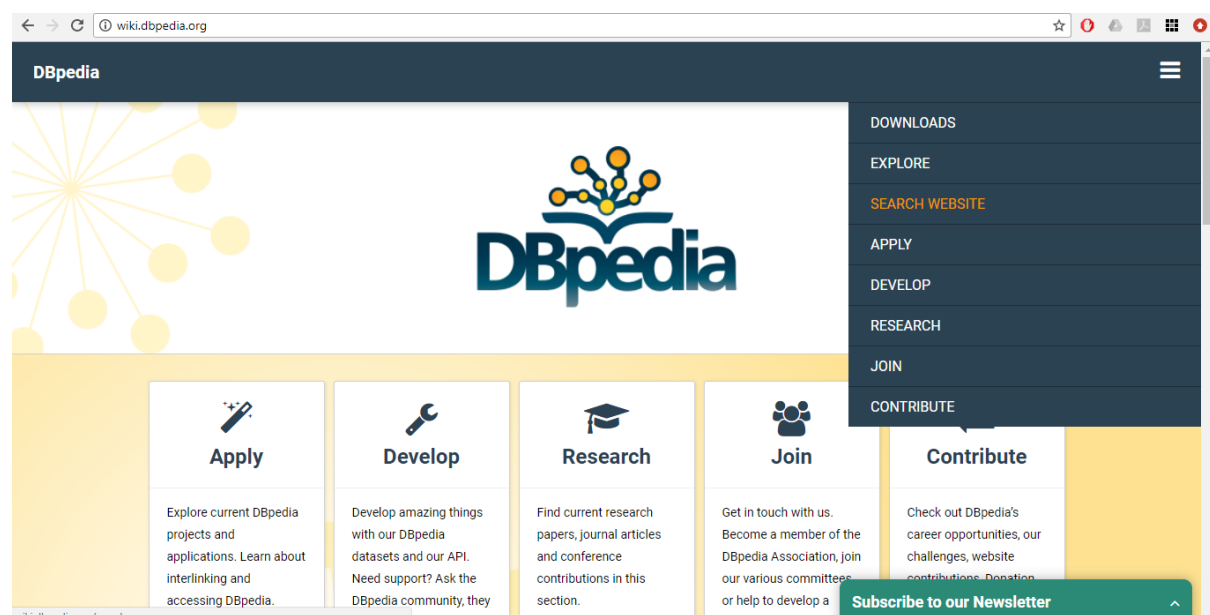


Figure 4 - DBpedia home page

The homepage looks very modern and everything is clear and easy to read but there is no way of searching its data directly from the page. Instead, the user must navigate to 'Search Website' in the drop-down menu in the top right.

Figure 4 shows the result when I navigated to this page to use it to search for a famous musical artist.

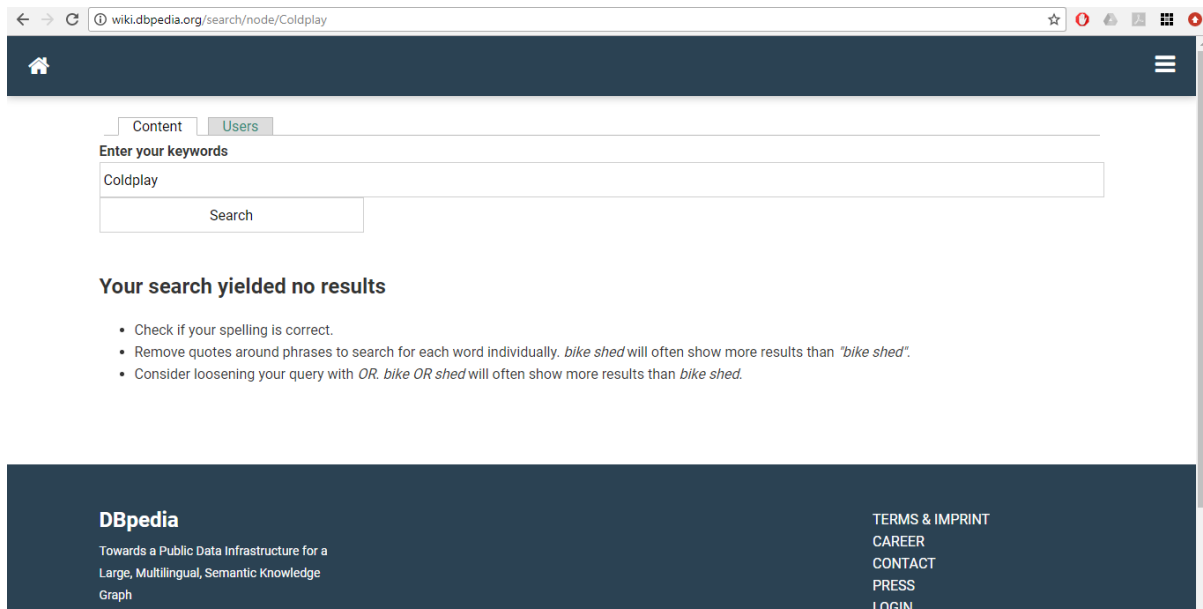


Figure 5 - DBpedia search page

From trying other search queries, it became apparent that there were no results because this search functionality does not query the dataset. It instead returns articles and DBpedia information pages related to your query.

Further investigation revealed that DBpedia has a 'Lookup' API for finding URIs by related keywords, but this service requires an installation and only queries data from the DBpedia 3.8 release, which was published in 2015. [23]

It turned out that the best way to query DBpedia to simply view information about a resource was via Google. For example, querying Google with 'Coldplay DBpedia' returned their resource page.

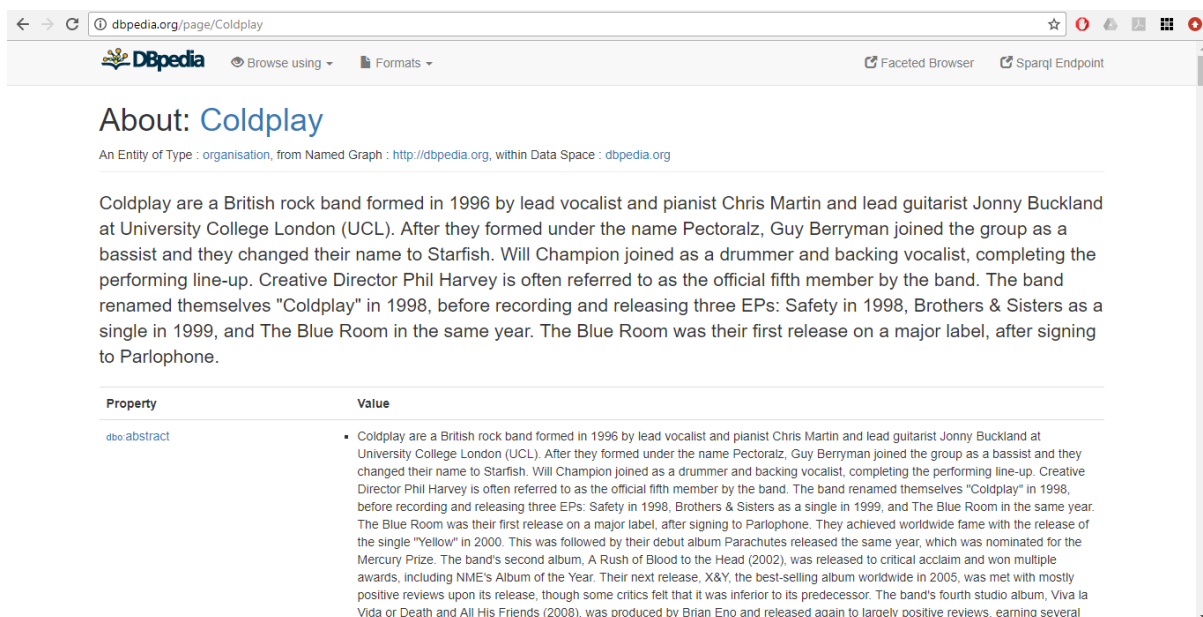


Figure 6 - Example of a DBpedia resource represented as an HTML page on the site

These resource pages can be very long and take a long time to scroll through to find desired information as they are simply a long list of Properties and their Values. Some of the Values are literals and therefore static text whereas others are links to other DBpedia resources or classes. The top right of DBpedia resource pages such as the one shown in [Figure 5](#) have a link to a SPARQL endpoint for querying the whole dataset.

← → ↻ dbpedia.org/sparql/ ☆ 🔍 📄 🗄️

Virtuoso SPARQL Query Editor [About](#) | [Namespace Prefixes](#) | [Inference rules](#) | [RDF views](#) | [SPARQL](#)

Default Data Set Name (Graph IRI)

Query Text  

```
select distinct ?Concept where {[ ] a ?Concept} LIMIT 100
```

(Security restrictions of this server do not allow you to retrieve remote RDF data, see [details](#).)

Results Format:

Execution timeout:  milliseconds (values less than 1000 are ignored)

Options:  
☒ Strict checking of void variables  
☐ Log debug info at the end of output (has no effect on some queries and output formats)  
☐ Generate SPARQL compilation report (instead of executing the query)

(The result can only be sent back to browser, not saved on the server, see [details](#).)

Copyright © 2018 [OpenLink Software](#)  
 Virtuoso version 07.20.3228 on Linux (686-generic-linux-glibc25-64), Single Server Edition

Figure 7 - Interface for SPARQL endpoint hosted by DBpedia

Although very powerful, this interface is not very easy to understand for beginners. Here is an example SPARQL query to retrieve members of the band Coldplay:

```
PREFIX dbo: http://dbpedia.org/ontology/
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?label WHERE {
  <http://dbpedia.org/resource/Coldplay> dbo:bandMember ?member.
  ?member rdfs:label ?label.
  filter langMatches(lang(?label),"en") }
```

Figure 8 - An example SPARQL query

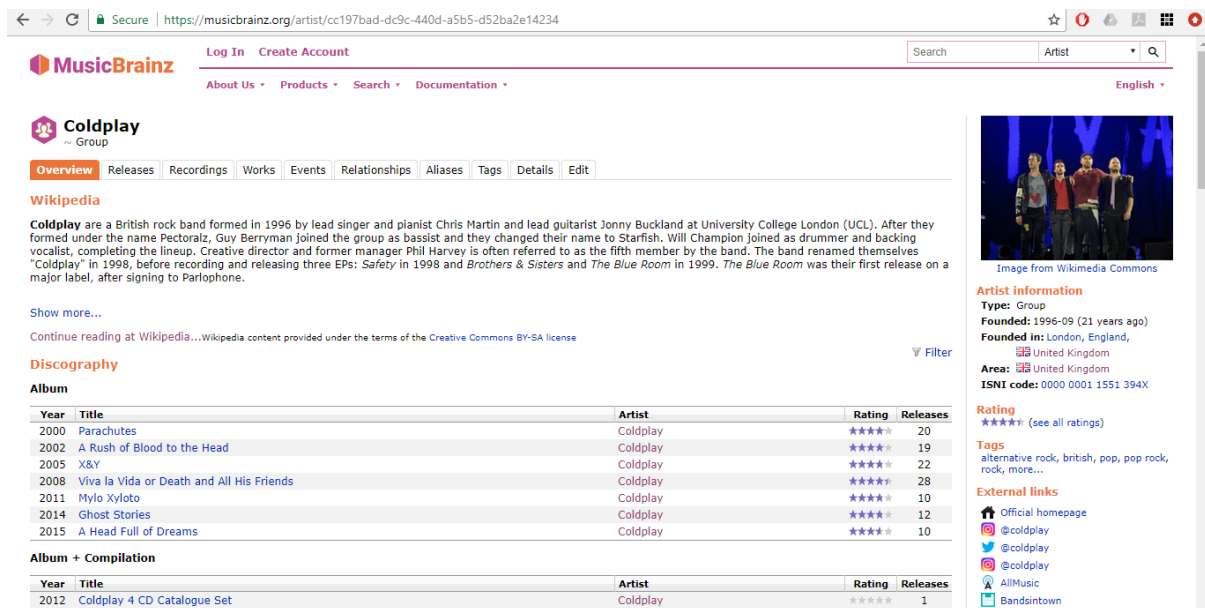
To write this query the user must:

- Assign prefixes which direct to the correct vocabulary for the `bandMember` and `label` properties.
- Understand how to create a variable in a query using a `?`.
- Know how to apply a filter to ensure the `SELECT` language is returned in a language they understand.

It is endpoints like Figure 7 that my application uses to access Linked Data, but the users of the application I developed do not need to write any SPARQL queries themselves. The queries required to find the information that users value most is already written and implemented by the back-end of the application.

## 2.6.2 Querying MusicBrainz

Searching for information on MusicBrainz is much easier than DBpedia as all MusicBrainz pages contain a search bar in the right-hand corner, which makes the site very easy to query.



The screenshot shows the MusicBrainz artist page for Coldplay. The page layout includes a navigation bar at the top with links for Log In, Create Account, and a search bar. Below the navigation bar, the artist's name 'Coldplay' is displayed with a group icon. A tabbed interface shows 'Overview' as the selected tab, with other tabs for Releases, Recordings, Works, Events, Relationships, Aliases, Tags, Details, and Edit. A Wikipedia summary of the band is provided, followed by a 'Show more...' link. Below this is a 'Discography' section with a table of albums. To the right, 'Artist information' is displayed, including the band's type, founding date, location, area, and ISNI code. A 'Rating' section shows a 4.5-star average, and 'Tags' list genres like alternative rock and pop. 'External links' include the official homepage and social media profiles.

**Wikipedia**

**Coldplay** are a British rock band formed in 1996 by lead singer and pianist Chris Martin and lead guitarist Jonny Buckland at University College London (UCL). After they formed under the name Pectoralz, Guy Berryman joined the group as bassist and they changed their name to Starfish. Will Champion joined as drummer and backing vocalist, completing the lineup. Creative director and former manager Phil Harvey is often referred to as the fifth member by the band. The band renamed themselves "Coldplay" in 1996, before recording and releasing three EPs: *Safety* in 1998 and *Brothers & Sisters* and *The Blue Room* in 1999. *The Blue Room* was their first release on a major label, after signing to Parlophone.

[Show more...](#)

[Continue reading at Wikipedia...](#) Wikipedia content provided under the terms of the [Creative Commons BY-SA license](#)

**Discography**

**Album**

Year	Title	Artist	Rating	Releases
2000	Parachutes	Coldplay	★★★★★	20
2002	A Rush of Blood to the Head	Coldplay	★★★★★	19
2005	X&Y	Coldplay	★★★★★	22
2008	Viva la Vida or Death and All His Friends	Coldplay	★★★★★	28
2011	Mylo Xyloto	Coldplay	★★★★★	10
2014	Ghost Stories	Coldplay	★★★★★	12
2015	A Head Full of Dreams	Coldplay	★★★★★	10

**Album + Compilation**

Year	Title	Artist	Rating	Releases
2012	Coldplay 4 CD Catalogue Set	Coldplay	★★★★★	1

**Artist information**

Type: Group  
 Founded: 1996-09 (21 years ago)  
 Founded in: London, England,  
 United Kingdom  
 Area: United Kingdom  
 ISNI code: 0000 0001 1551 394X

**Rating**  
 ★★★★★ (see all ratings)

**Tags**  
 alternative rock, british, pop, pop rock, rock, more...

**External links**

- Official homepage
- @coldplay
- @coldplay
- @coldplay
- AllMusic
- Bandaintown

Figure 9 - A MusicBrainz artist information page

When an album is first clicked on MusicBrainz, the user is not immediately taken to an album information page, they must first select a specific release as shown by Figure 10.



**MusicBrainz** Log In Create Account

Search Artist

About Us Products Search Documentation English

**X&Y**  
~ Release group by Coldplay

Overview Allias Tags Details Edit

**Wikipedia**

**X&Y** (stylized as **X & Y**) is the third studio album by the British rock band Coldplay. It was released on 6 June 2005 by Parlophone in the United Kingdom, and a day later by Capitol Records in the United States. The album was produced by Coldplay and producer Danton Supple. It is noted for its troubled and urgent development, with producer Ken Nelson having originally been tasked with producing much of the album; however, many songs written during his sessions were discarded owing to the band's dissatisfaction with them. The album's cover art is a combination of colours and blocks, which is a representation of the Baudot code.

The album contains twelve tracks and an additional hidden track, "Til Kingdom Come". It is omitted from the track listing on the album sleeve, but listed as "+" on the disc

Show more...

Continue reading at Wikipedia...Wikipedia content provided under the terms of the Creative Commons BY-SA license

**Album**

Release	Format	Tracks	Date	Country	Label	Catalog#	Barcode
<b>X&amp;Y</b>	CD	13	2005-06-01	JP	Parlophone	TOCP-56370	4988005827929
<b>X&amp;Y</b>	Copy Control CD	13	2005-06-03	XE	Parlophone	00946 311280 2 8	0094631128028
<b>X&amp;Y</b>	Copy Control CD	13	2005-06-06	AU	Parlophone	094631128028	094631128028
<b>X&amp;Y</b>	Digital Media	13	2005-06-06	BR	Parlophone	[none]	724347478659
<b>X&amp;Y</b>	CD	13	2005-06-06	US	Parlophone	0724347478628	724347478628
<b>X&amp;Y</b>	CD	13	2005-06-07	CA	EMI Music Canada, Parlophone	09463 11280 2 8	094631128028
<b>X&amp;Y</b>	CD	13	2005-06-07	US	Capitol Records (imprint of Capitol Records, Inc.)	74786	724347478628
<b>X&amp;Y</b>	CD	13	2005-09-30	JP	Parlophone	TOCP-66465	4988006834934

**Release group information**

Artist: Coldplay  
Type: Album

Rating: ★★★★★ (see all ratings)

Tags: rock, alternative and punk, alternative rock, pop rock, acoustic, more...

External links: AllMusic, BBC Music, Discogs, en: X&Y, LyricWiki, MusicMoz, Q158738, Rateyourmusic

**X&Y**  
~ Release by Coldplay (see all versions of this release, 22 available)

Overview Disc IDs Cover Art Allias Tags Details Edit

**Annotation**

Til Kingdom Come, is a hidden track.

Barcodes **4988006840119** and **9787799624266** are for a special tour edition that include a bonus disc.

Annotation last modified on 2008-12-01 18:45 UTC.

**Tracklist**

Display Credits at Bottom

#	Title	Rating	Length
1	Square One strings: Susan Dench, Richard George (violinist), Peter Lale, Laura Melhuish, Audrey Riley and Chris Tombling strings arranger: Audrey Riley guitar: Matt McGinn (UK guitarist for Rosita) mastering: Chris Athens and George Marino mixer: Michael H. Brauer (engineer) producer: Coldplay and Danton Supple phonographic copyright by: Parlophone Records Ltd. (not for release label use) a Warner Music Group company) (2007) recording of: Square One writer: Guy Berryman, Jonny Buckland, Will Champion and Chris Martin (lead singer of Coldplay)	★★★★★ 4:47	
2	What If strings: Susan Dench, Richard George (violinist), Peter Lale, Laura Melhuish, Audrey Riley and Chris Tombling strings arranger: Audrey Riley mastering: Chris Athens and George Marino mixer: Michael H. Brauer (engineer) producer: Coldplay and Danton Supple phonographic copyright by: Parlophone Records Ltd. (not for release label use) a Warner Music Group company) (2007) sampled by: What If We Cry? by Mick Boogie & Terry Urban recording of: What If writer: Guy Berryman, Jonny Buckland, Will Champion and Chris Martin (lead singer of Coldplay)	★★★★★ 4:57	

**Release information**

Barcode: 724347478628  
Format: CD  
Length: 1:02:36

**Additional details**

Type: Album  
Packaging: Jewel Case  
Status: Official  
Language: English  
Script: Latin  
Data Quality: Normal

**Labels**

- Parlophone 0724347478628

**Release events**

- United Kingdom 2005-06-06

**Release group rating**

Figure 10 - A MusicBrainz release-list page (Top) and an individual release information page (Bottom)

For the application I was developing, I wanted to remove this extra step for the user by just showing them the track listing for the initial release of the album. I wanted my album information pages to be an amalgamation of the two pages in Figure 10, with a track list, initial release date, tags (or genres) and description.



## 3 Specification and Design

### 3.1 Requirements

This section is a summary of the requirements identified in the software requirements specification document I created in the early stages of the project (see Appendix A). Any requirements added after spending some time exploring the application's three data sources are marked with an asterisk.

#### 3.1.1 Functional Requirements

Artist information pages must contain:

- A brief description of the artist.
- All albums released by the artist.
- List of band members and what instrument they play.
- List of associated genres.
- Hometown if available.
- Link to the artist's official site if available.

Album information pages must contain:

- A brief description of the album
- The album's year of release
- The producer(s) of the album
- Highest sales award (e.g. Platinum, Gold)
- Genres associated with the album

#### 3.1.2 Non-functional requirements

- All web pages contain a search bar for finding artists, albums or songs.
- Web pages should look modern and be easy to navigate/understand.
- Web pages should cater for accessibility issues (e.g. visual impairment, colour blindness)
- Users should have to navigate through less pages than they do on MusicBrainz to find certain information.
- Artist pages:
  - Include an image of the artist.
  - List some of their highest rated tracks according to ratings on MusicBrainz.
  - Links to external social media sites belonging to the artist: Facebook, Twitter, Instagram and YouTube
  - YouTube videos from an artist's channel are displayed on the artist pages.
  - A link to the artist's corresponding Bandsintown page so that users can see if the artist is on tour near them
  - A link to the artist's corresponding setlist.fm page so that users can view setlists of the artist's past performances.
- Album pages:
  - Album artwork is displayed next to the album name.
  - Contain a list of tracks from the album.

Studies into how people search for classical music helped me to decide what information was essential for my application. Such studies found that there were six common ‘access points’ between participants, which were deployed over 75% of the time. These were composer, title, performer, genre, opus/thematic number and instrumentation. If you roughly convert these findings into pop music entities for my application you would have producer, album name, artist and instrumentation. [24]

### 3.2 Entity Relationship Diagram

With the applications requirements defined I had a clear idea of what information needed to be included on the application’s web pages. Figure 11 is an Entity Relationship Diagram (ERD) that shows the three types of information pages for the application. Originally only Artist and Album pages were planned but I wanted to also include Genre pages. Spending time learning how to use the SPARQL wrapper package revealed it would not be too time-consuming to implement queries to retrieve Genre description and lists of sub genres. These descriptions can be quite long on DBpedia, so it made sense to give Genres their own pages.

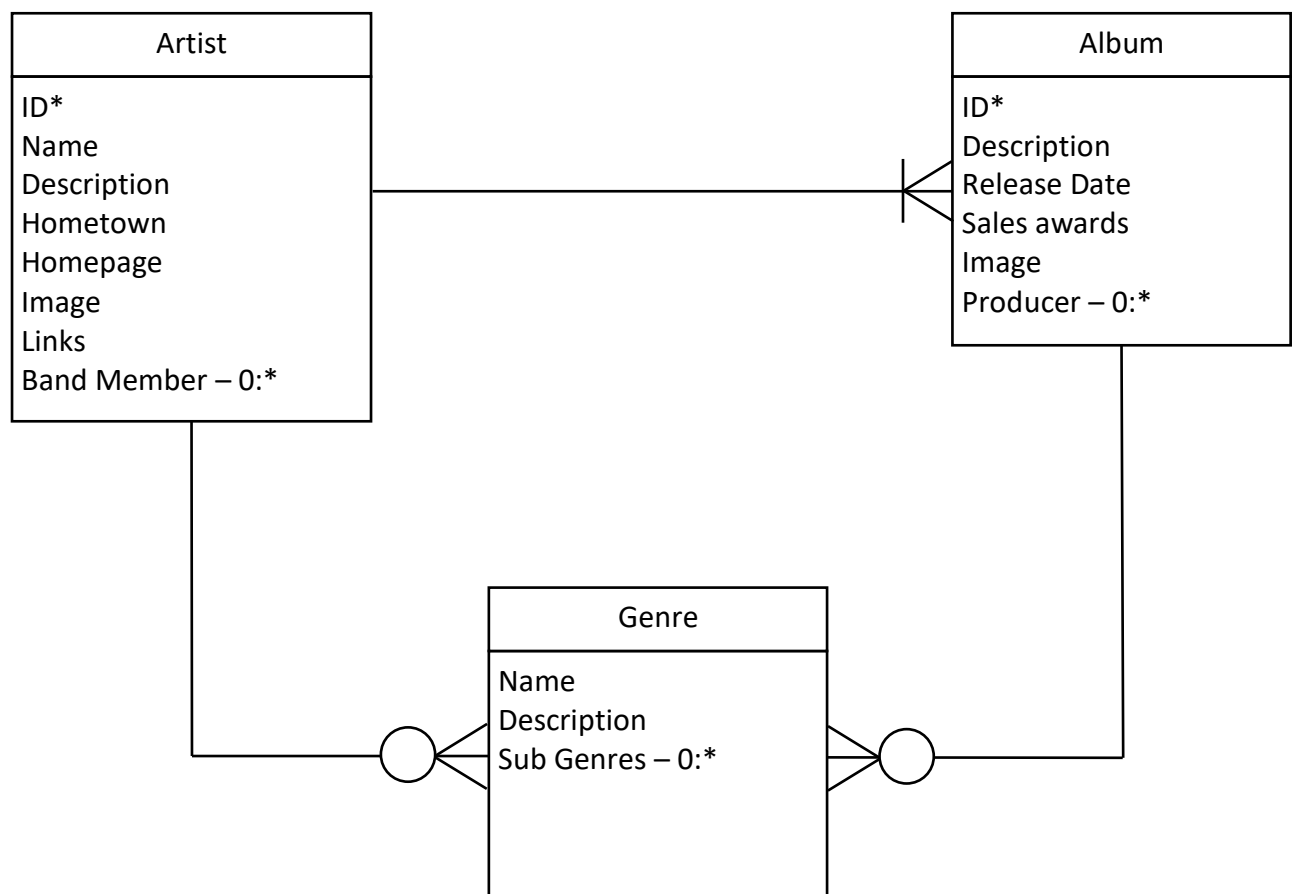


Figure 11 - Entity Relationship Diagram (ERD) for my application

The diagram indicates that artist entities can have one-to-many album entities associated with them. Both of these entities can then have zero-to-many genres associated with them.

### 3.3 Data Collection System

The back-end of the system used the MBAPI and a selection of SPARQL queries to collect relevant music-related data. Figure 12 is a flow diagram which shows the basic steps taken by the application to obtain and display information. As the MBAPI returns information in lists of dictionaries, it was logical to store information extracted from these dictionaries within the application's own dictionary objects. The application's own dictionaries were then passed to HTML templates which could use its values as content when rendering the web page. Django's flow controls were another reason for sticking to dictionaries as keys can be specifically referenced, and lists can be iterated using 'if' statements and 'for' loops.

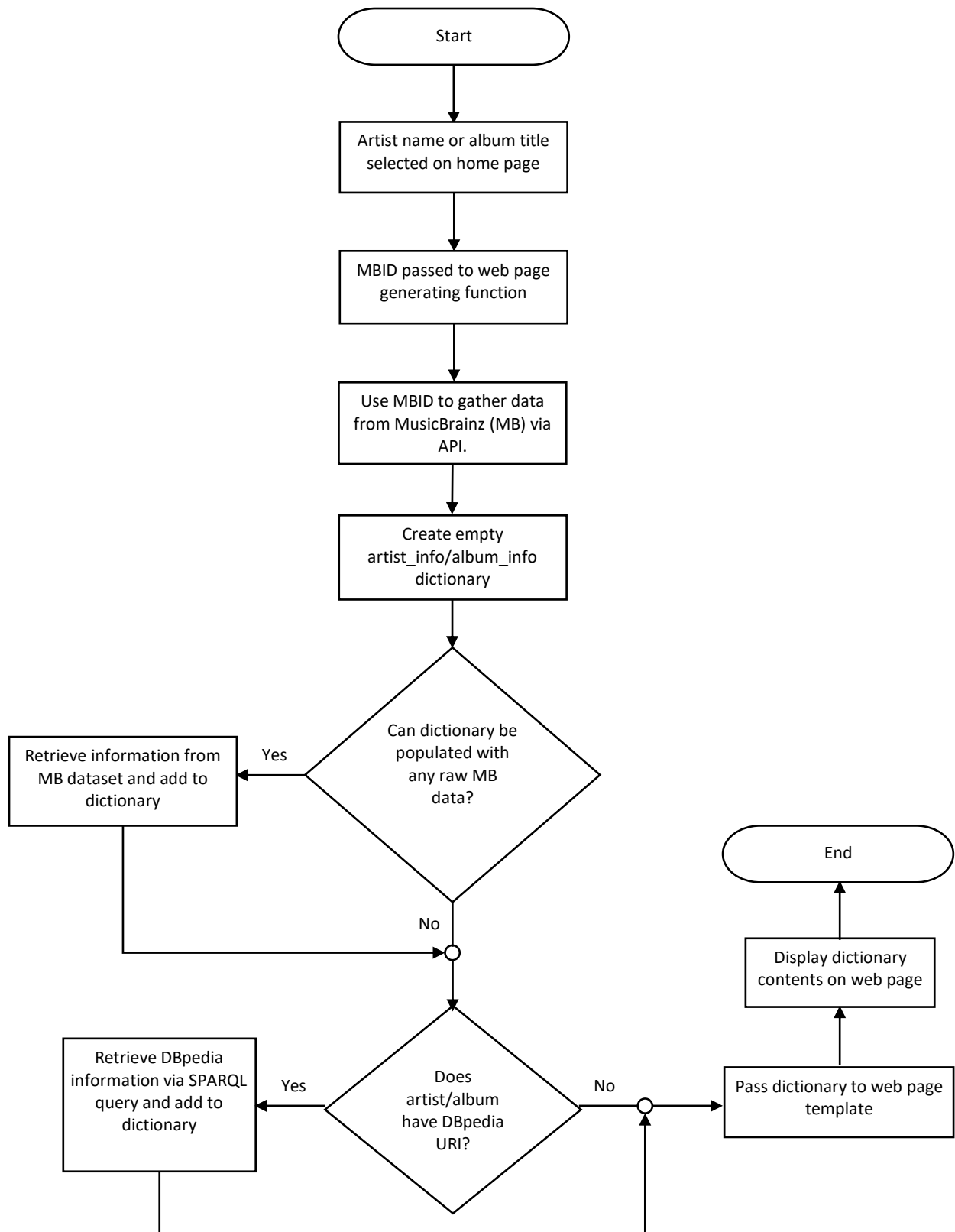


Figure 12 - Flow diagram to illustrate the procedures taken by the application's data collection system

### 3.4 User interface

In Section 2.6.1 Querying DBpedia an example SPARQL query shows how a user needs to have specialist knowledge to retrieve information. They also need to use Google search to check if resources and properties even exist for certain artists. My application was designed to make it easier to retrieve information from DBpedia by focusing on three of Jakob Nielsen's Usability Principles.

Firstly, the system matches the real world in that it uses words, phrases and concepts familiar to the user. Information on the page is clearly labelled with headings and the user does not need to know any SPARQL. The page layout also follows common conventions which I will elude to throughout this section as I describe the layout I chose for each web page. Secondly, the application follows consistency and standards as each page has a very similar layout, so the user can learn to navigate the application very quickly. Finally, the application has an aesthetic and minimalist design. It is more colourful than MusicBrainz and, unlike DBpedia, has images and icons.

The web application has four types of pages:

- Home page
- Artist pages
- Album pages
- Genre pages

#### 3.4.1 Wireframes

I created wireframe designs for the application's web pages before beginning the implementation as I wanted a clear vision of what I wanted to create before using CSS.

##### Home Page



Figure 13 - Home page of the application

The site's homepage is an example of minimalist design, which makes it easy to understand for the user and allows for quick loading times. When first visiting the site, the white space

under the blue banner will be blank. A list of search results will only be returned once text has been typed into the search bar and the search icon clicked/enter key pressed.

A search bar will be the only interactive element on the home page because I decided it would be acceptable for my website to rely on 'known-item searching' as a means for users to find information. This is any search where the user knows that an item exists and maybe already has some information about it [4]. I thought this would be the best method as 'anecdotal evidence supports the idea that known item searching is more common for music materials than books'. Two studies observing University students by David M. King in his research on user search strategies for finding music materials showed that over 65% of participants used the name of a piece of music's author as a search query [25]. Another study of Google Answers queries relating to music found that most queries were concerned with finding musical works followed by queries for identifying artists. Therefore, I decided that my application search bar should just offer options to search for either artists or albums. [4]

### Artist Pages

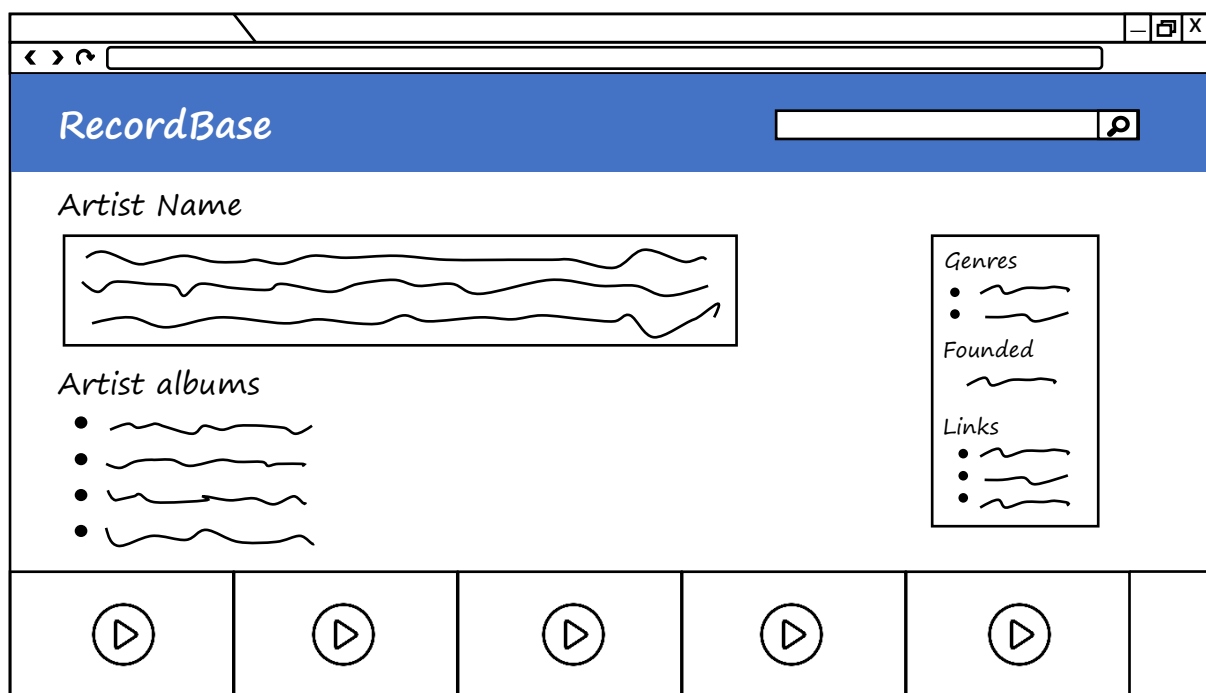


Figure 14 - Artist information page from the application

The banner remains the same as it does on the home page to maintain a consistent theme across the site. The first text block under the Artist's name is a description. The bullet point list underneath it is a list of the artist's Album releases. The side bar at the right of the screen will display a list of genres associated with the artist, where they originated and a list of links to some of most visited web pages related to music artists.

This page layout is following conventions that a user will recognise facilitating ease of use. The placement of the description and sidebar is consistent with sites like Wikipedia and MusicBrainz, but my application is more simplistic and eye catching with larger text than MusicBrainz pages which will help users with visual impairments.

The Genres section in the side bar displays genres listed on an artist's DBpedia page. In comparison to the process of obtaining this information detailed in Section 2.6.1, viewing this information is much easier in my application as no time needs to be spent writing a SPARQL query. The user just searches for an artist, clicks their name on the homepage and then they are directed to this page where genres are listed. If they wanted to view such information on a DBpedia resource page itself instead of using a SPARQL query, they would first have to use Google as DBpedia itself does not facilitate searching for resources.

If there is enough time for implementation, a carousel at the bottom of the page will automatically scroll through a selection of videos uploaded to YouTube by the artist, so long as a YouTube channel for that artist can be found. One of the great aspects of the BBC Music site is how visual it is, they have many audio clips and videos on their artist information pages. I therefore want to follow this idea and include the carousel to make my application look more modern and eye-catching than MusicBrainz but also offer more discography information than the BBC Music site. Furthermore, my application would be offering video content the BBC cannot.

(Unfortunately, I did not have enough time to implement this feature in the application itself. See Section 5.3.2 Non-functional requirements for information regarding potential implementation).

#### *Album page*

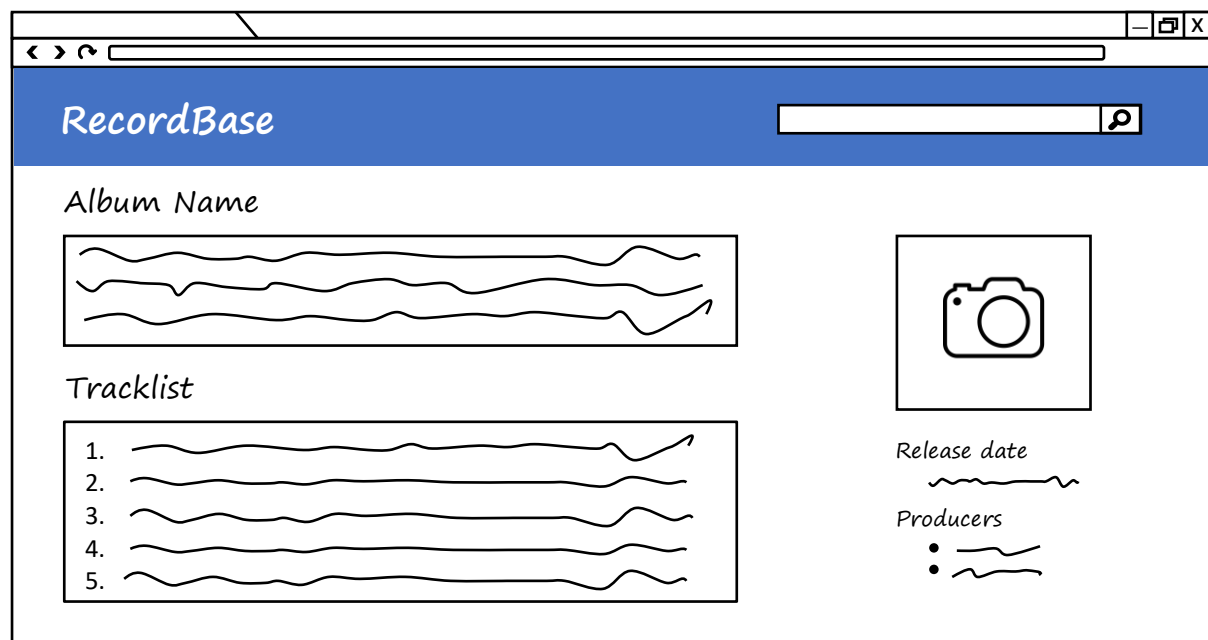


Figure 15 - Album information page from the application

Album pages will be similar to Artist pages in that they have the banner, description, and side bar content. Underneath the descriptive paragraph there will be a numbered track list of the album. If there is time to implement, there will be a photo of the album artwork on the right-hand side of the page. The rest of the right-hand side will consist of a release date for the piece of work, highest sales award and, like Artist pages, the genres associated with the release.

### Genre pages

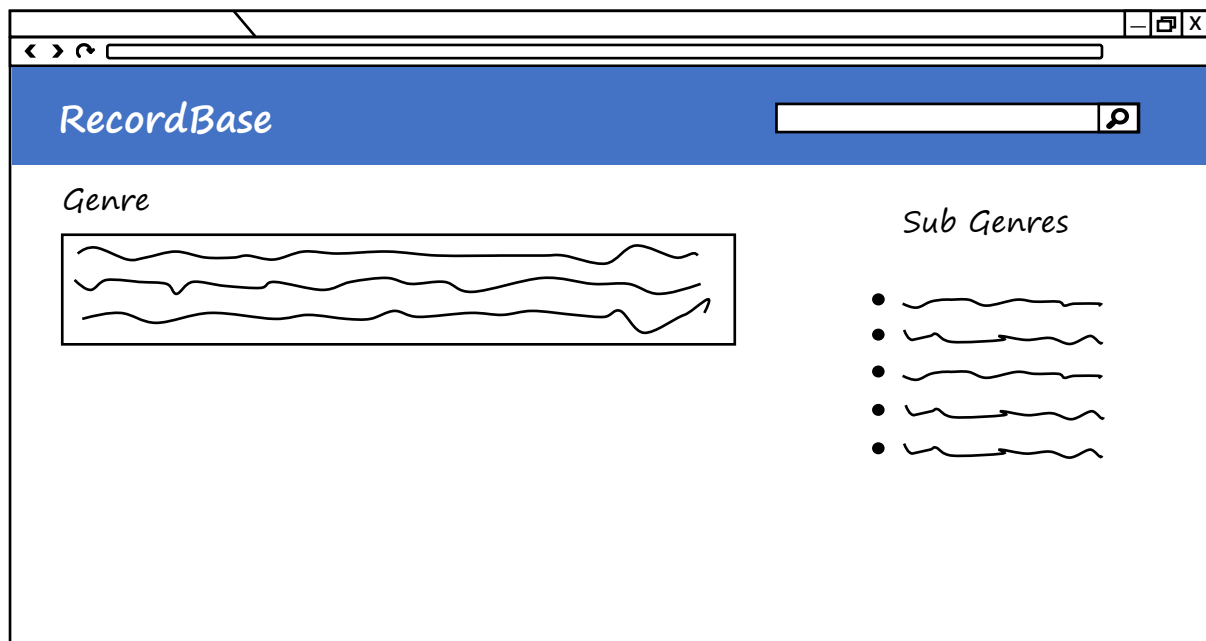


Figure 16 - Genre information page from the application

Genre pages will include a description like the other pages and a list of sub genres on the right-hand side if applicable. The presence of the search bar on every page of the application provides efficiency of use. It allows experienced users to quickly locate artist/album pages they already know exist. Being consistent throughout the platform is also beneficial to users as it removes doubt and makes the application easier to use.

#### 3.4.3 Choice of External Links for Application

MusicBrainz offers a selection of 'External Links' on the right-hand side of their artist web pages. Although this is useful I feel that they provide too many links and that users are not going to spend time looking through all of them. I therefore decided to try and determine which of these links were the most useful and only include those.

I wanted one of the links to be to a song lyrics website as studies show that people rank lyrics as the second most-valued item of information after a song's title when they are seeking music information [4]. Of the numerous lyrics sites available, I selected Genius.com for several reasons. Not only is it a platform for song lyrics but also music news. It is contributed to by its community and has over 100 million monthly visitors [26]. Furthermore, unlike its biggest competitors (AZ Lyrics, Lyrics.com, MetroLyrics), lyrics for a song can be clicked to reveal more information and context such as explanations or interpretations of lyrics.

Due to the popularity of social media sites, links to these were also crucial. Of all the links from MusicBrainz that I could use, the following four sites had the greatest number of users, from highest number of users to the least: Facebook, YouTube, Instagram and Twitter. As of April 2018, all these networks combined had over 330 million users, with Facebook having around 2,234 million users [27]. I therefore included links to these four sites on each artist's web page along with links to their own website and their Soundcloud page if they had one, as it is the 'world's largest music and audio platform' for artists of any level of fame to freely share their music.



The last two links that I thought were very important to include were to [setlist.fm](https://setlist.fm) and [songkick.com](https://songkick.com). The former is a collection of community-curated setlists from concerts, so that attendees can find out exactly what songs were played at them. People can also use it to get a rough idea of what an act is going to play before they go and see them. The latter, Songkick, is a service which helps its 15 million worldwide users to find concerts near them that they might be interested in and alert them when one of their favourite artists will be playing near their location.

I feel that all these links will provide users of the application with sufficient information, and unlike the 'External Sites' lists on MusicBrainz, sites that offer the same information as another are not included. Streamlining the amount of options for the user makes the application easier to use as it requires less decision making.

## 4 Implementation

This section will cover the finer details of the application's development. Any noteworthy discoveries or problems encountered during the implementation of the application's features will be discussed. Where appropriate, sections of code will be referenced to illustrate more clearly how solutions are working.

I will first give an overview of the application's general implementation, describing where data for each region of the page was obtained and also demonstrating via screenshots of the application, how it is easier to search for an artist and reach their information page than it is on DBpedia or MusicBrainz. I will then move on to describe in greater detail, aspects of the system which were harder to implement, explaining how I overcame the problems.

(Please note that the text file 'File Locations' in Archive 1 details the folders in which views.py and HTML files referred to frequently in this section are stored)

### 4.1 Choice of platform

As mentioned in section 2.5, Django was used as the application's platform. Even though this choice was time consuming because I had to learn from scratch how to use Django, it was worthwhile as it provided more flexibility during development. The web server Django allows you to set up on your own machine offers clarity when pages fail to render correctly due to errors in code. Specific Python error messages are shown, indicating the line numbers of failing code and the state of variables at various points of program execution. When beginning the project on the School's own web servers there was none of this transparency when it came to error handling, so if I had continued to use it for the rest of the project debugging would have been incredibly difficult and time-consuming. Django also has extensive documentation which was useful in development.

### 4.3 Home Page

The first page of the website would have no other features than search. To fulfil functional requirements, the application needed to return a list of artists or albums based on the query text string entered by the user. The ability to search for these two entities was the very minimum.

Figure 17 shows how the application looks when the user first lands on the home page:

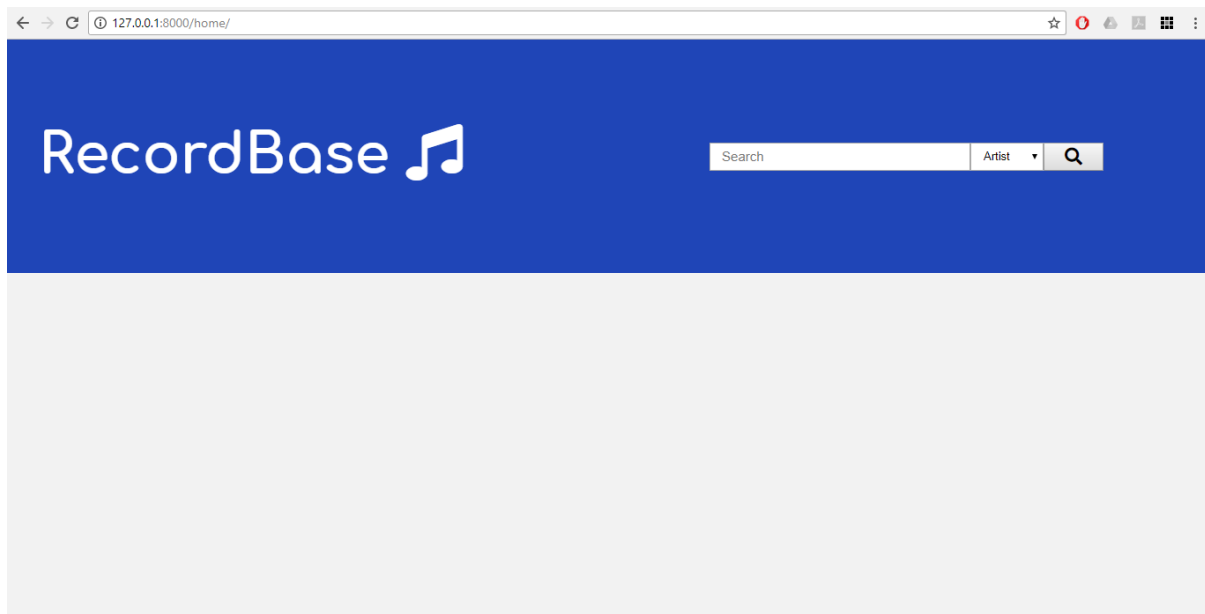


Figure 17 - Screenshot of the application's home page with no search results

The user's attention will immediately be drawn to the search bar as it is the only element on the page that can be interacted with.

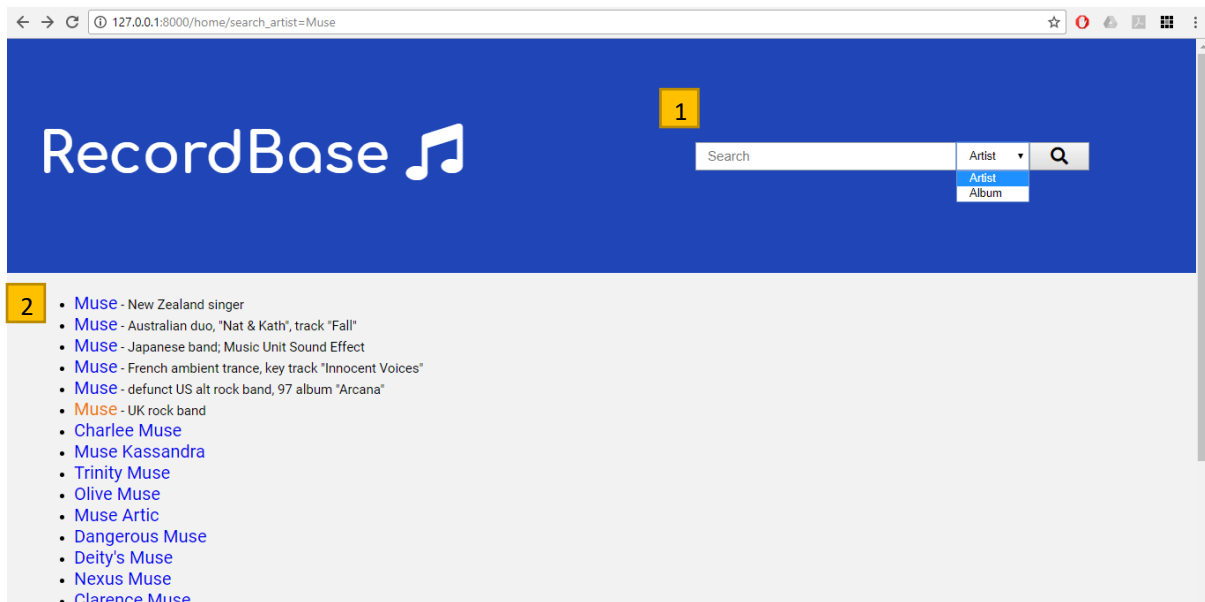


Figure 18 - Screenshot of the application's home page once a search query has been submitted

1. The search feature was implemented using MBAPI functions which returned either a list of artists or release-groups. The drop-down menu in the search bar offers the user a choice between 'Artist' or 'Album'. In this example the query text entered was 'Muse' and the option 'Artist' selected. This is reflected in the end of the URL. Notice how 'search\_artist=Muse' has been appended to the original URL shown in Figure 17. Both the phrases 'artist' and 'Muse' have been passed to views.py as 'query\_type' and 'query' parameters respectively for the search function. If the user had selected the 'Album' option from the drop-down menu, then the URL would

have been appended with 'search\_album=Muse' in this example and 'album' passed as the 'query\_type' argument.

Search functionality needed to be achieved using the MBAPI and not SPARQL queries to DBpedia, as MusicBrainz unique IDs (MBIDs) needed to be returned. This is because the various get methods offered by the MBAPI, which are necessary for implementing other parts of the application, rely on MBIDs to be used as function parameters.

2. When the user clicks on the search icon or hits the enter key, a list of results based on their query is returned. In this example the artist's names are on the left, clearly marked as hyperlinks in blue text which turn orange if hovered over. When applicable, the artist's name is followed by a short statement which may help the user disambiguate results. The MBAPI search function used to obtain search results, includes such statements in the dictionary it returns if there is one associated with such artist. Album searches return similar lists except there is not a disambiguation statement. Instead, every clickable album name is followed by the primary release artist of this album. This means that if multiple artists worked on a project, only the one returned first by MusicBrainz is listed.

#### 4.4 Artist Pages

Once the user has clicked on the artist they wish to view from the home page they will be taken to a page displaying information about that artist. Note that in this example the original URL has been appended with 'artist/9c9f1380-2516-4fc9-a3e6-f9f61941d090'. The last part of this URL is this artist's MBID. The url.py file passes this MBID as an argument to the *artist\_page* function in views.py.

The first action taken by the *artist\_page* function uses the MBAPI function *get\_artist\_by\_id* to gather a large information set about that artist from MusicBrainz. This data is returned as a dictionary, but not all the data within the dictionary is useful or necessary for the user, so a new empty dictionary (*artist\_info*) is created by the function. It is this dictionary which is eventually passed to the Django template *artist.html* when the *artist\_page* function is completed. The rest of the *artist\_page* function assigns data for the web page to keys in the *artist\_info* dictionary, using various get methods devised in the rest of views.py

The list below Figure 19 details the data sources for elements of an artist information web page:

The screenshot shows the RecordBase website's artist page for the band Muse. The page is titled 'Muse' and includes a detailed description of the band's history and discography. A list of albums is provided in reverse chronological order, with each album title linked to its own information page. The page also displays the band's genres, their origin (Devon, GB), and the names of the band members. A footer section contains various social media and streaming service links. An 'Export RDF' button is located near the top right of the main content area.

Figure 19 - Screenshot of an artist information page

1. A description of the artist is obtained from DBpedia via SPARQL query using the *dbo:abstract* property. The *get\_description* function in views.py implements this. There were significant challenges implementing this feature due to use of the correct DBpedia URLs and ambiguity between different entities. Section 4.6.5 Dealing with ambiguity when getting descriptions describes this in more detail.
2. A list of albums released by the artist is shown in reverse chronological order, with the year of initial release displayed next to the album title. Each album title is a hyperlink to an information page about the album in question. The list of albums is obtained using the *get\_albums* function in views.py.

3. An image of the artist is obtained from DBpedia via SPARQL query using the *dbo:thumbnail* property. The *get\_artist\_image* function in *views.py* implements this.
4. A list of genres assigned to the artist is obtained using the *get\_tags* and *get\_dbpedia\_genres* functions. This process means that data is retrieved from both MusicBrainz and DBpedia. Aggregating the data from these sources will be discussed in more detail in Section 4.6.2 Handling Genres.
5. In this area of the page, the geographical location from which the artist originated is displayed. This information is obtained from MusicBrainz. The *get\_location* function which returns this information checks the dictionary keys *'begin-area'* and *'country'* are in the MusicBrainz data, returning them if they are present.
6. If the artist is a group, a list of band members is obtained using a SPARQL query to DBpedia. This is described in detail in Section 4.6.1 Retrieving Band Members, as inconsistency regarding data and DBpedia ontology properties made retrieving this information more complicated than initially foreseen.
7. This section at the bottom of each artist page contains a selection of links to popular social media and music related information sites. Each link has a 'font-awesome' icon above it which is an official logo of the site the link goes to (provided font-awesome had a logo). To give the web application a more modern feel, I thought it was important to make it as pictorial and graphical as possible. The font-awesome icons helped me to achieve this. When people think of sites like Facebook and Twitter, they are likely to think of their logos first due to the recognisability of their brands. The related links on artist web pages were therefore the ideal place to use such logos on the web application. The size of the icons and horizontal display of the external links list makes them stand-out more than the MusicBrainz equivalents. The hyperlinks themselves were returned as part of the data from MusicBrainz. The *get\_links* function in *views.py* simply checks such a list is present and then assigns URLs and icons to a new dictionary for the HTML template.
8. The last feature implemented on the page is a 'Export RDF' button. When clicked this button allows the user to download a Turtle file which contains RDF triples related to the artist. Section 4.6.3 Creating RDF explains this functionality in more detail.

## 4.5 Album Pages

Album pages follow a very similar format to artist information pages, with the same top banner, title, main content and ‘side-bar’ providing extra information. The function *album\_page* which renders *album.html* with information also takes an MBID as a parameter. (URL ends with ‘album/6f151223-f3a3-3e57-810f-598f7897006c’ in Figure \* example). The MBID in the URL is again handled by *urls.py* which passes it to *views.py*.

The MBAPI function *get\_release\_group\_by\_id*, uses the MBID to retrieve all the information regarding the album in question. Like the *artist\_page* function, the rest of *album\_page* then creates and populates a dictionary to pass MusicBrainz and DBpedia data to *album.html*. However, *album\_page* is more complicated, as it contains four of its own functions for exclusively retrieving album information and it uses SPARQL to query the BBC Music site.

The numbered Figure 20 again corresponds to the list succeeding it:

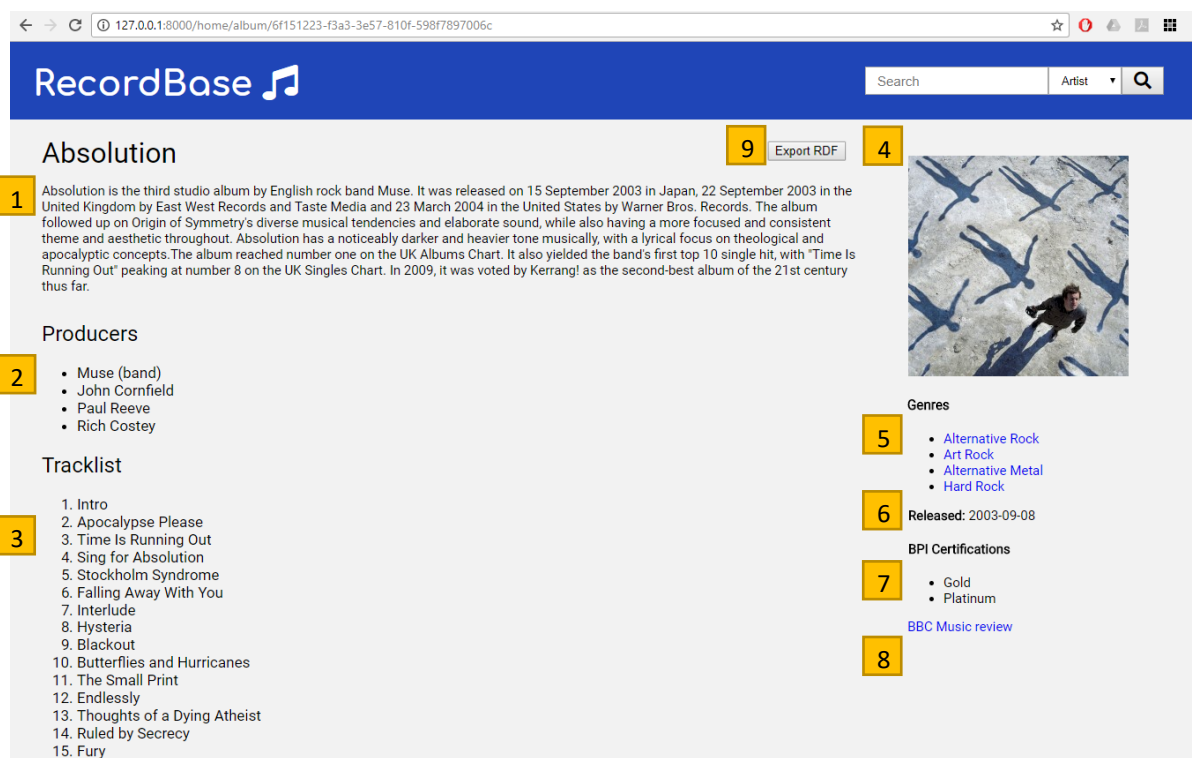


Figure 20 - Screenshot of an album information page

1. The description is obtained from the release’s DBpedia resource in exactly the same way as artist information pages.
2. A list of producers of the album is obtained using a SPARQL query, utilising *dbo/dbp:producer* and *rdfs:label* properties. The *get\_producers* function within *album\_page* implements this.
3. The tracklist of a release comes from MusicBrainz. This functionality is handled by the *get\_tracklist* function within *album\_page*. Section 4.6.4 Getting a track list covers obtaining tracklists in more detail as it was one of the most complicated parts of the application to implement and several methods of doing so were tested before settling on the final solution.
4. A MBAPI function call was used to get cover art for the release. DBpedia was only used to retrieve images for artists as it could not be done using the MBAPI. As a

result of using DBpedia for artists, images are not always available or cannot be rendered. However, MusicBrainz is much more likely to have a cover art image associated with a release so it is a much more reliable way of getting an image. However, the function call assigning the image to the *album\_info* dictionary is still encapsulated within a try/catch block to ensure the web page is still rendered properly if an image cannot be found (again, avoiding a `KeyError`).

5. The genres list on this page is similar to the tag list on artist information pages but this time, the list exclusively contains music genres and is fetched from DBpedia via SPARQL query. Again, Section 4.6.2 Handling Genres has more information.
6. The release date is simply the 'first-release-date' key entry from the returned MusicBrainz data.
7. This area is a list of sales awards achieved by the release. A SPARQL query fetches this information from DBpedia.
8. A SPARQL query is used to obtain the URL for a BBC Music album review of the release if there is one available. As BBC Music use the same MBID's as MusicBrainz, I could create a valid URI for the album's artist using their MBID. Incorporating this URI into a SPARQL query found in an OpenLink tutorial returns a list of URLs for albums by the artist [28]. Finally, my application checks if the title of the album the user is viewing was returned by the query and renders a hyperlink if required.
9. Finally, the album page 'Export RDF' button works in the same way as on artist information pages except the turtle file output contains different triples, specific to the album instead of the artist.

## 4.6 Notable sections

This part of the report will elaborate on some notable code functionality and challenges faced during implementation. Solutions to certain problems will be explained with reference to their viability and potential for extended application functionality.

### 4.6.1 Retrieving Band Members

Examining DBpedia pages from a few music groups made it apparent that a list of band members could be retrieved using SPARQL queries. I created a '*get\_band\_members*' method that took a DBpedia URI as its input parameter, using this URI in a SPARQL query which could retrieve the list of band members. I tested this function with a few different artists, and whilst it worked correctly for some, with others a blank list was returned. I first noticed this using the URI for The Beatles' DBpedia page. Investigation of this page revealed that the *dbo:bandMember* property that the SPARQL query the code was using, was not present. However, as The Beatles are no longer together, the property *dbo:formerBandMember* was in use instead. I therefore decided that I needed to modify my code so that both current and former band members could be obtained. Originally, I thought I would have to write a second SPARQL query for this. Although viable, this solution would have been inelegant as a large block of code would have required copying with only one variable being changed.

As string formatting was already being used in the Python code to insert DBpedia URIs into the SPARQL queries being implemented, I realised that I could use this feature again to change the DBpedia Ontology property of the original query. To do this I created a list containing the strings 'bandMember' and 'formerBandMember'. Then, the command to set the query was placed in a for loop, which iterated through this list of two strings:



```
relationships = ['bandMember','formerBandMember']

for item in relationships:
    dbpedia.setQuery("""
        %s

        SELECT ?label WHERE {
        <%s> dbo:%s ?member.
        ?member rdfs:label ?label.
        filter langMatches(lang(?label),"en") }

        """ % (prefixes,resource,item) )
```

Figure 21 - Code from `get_dbpedia_genres` function in `views.py`

The first `%s` in the query inserts all the prefixes I have used for SPARQL queries throughout `views.py`. I decided to make the prefixes a global variable so that they would not have to be duplicated for multiple queries. The second `%s` in the first line of the `WHERE` clause inserts the URI (`resource` variable) for the artist for which the application is collecting information for. The final `%s` inserts the current item from the list after the `dbo` prefix, creating either `dbo:bandMember` or `dbo:formerBandMember` depending on the stage of the loop.

The next step was to append all band members retrieved to a list. When doing this it was vital to keep track of which property was used to retrieve each result. This was achieved by storing each member in the list as a tuple, first entry being their name and the second 'bandMember' or 'formerBandMember'. Finally, this list of all members was then iterated through and sorted into two different lists of either current members or past members based on the second tuple value of each entry.

Once passing the lists of current and past members to `artist.html`, it was important to make sure that only the relevant headings were shown. For instance, if an artist information page had a 'Current Members' heading with several names below it followed by a former member it would be confusing for the user. Fortunately, Django includes flow control options such as 'if' statements and 'for' loops. By using 'if' statements in the artist HTML template, headings and lists for current or former members would only be displayed on the page if the lists passed to the template from the back-end Python code were not empty. The fact that Django incorporates the flow controls was another reason why it was better to use instead of a traditional CGI-based platform.

Although the code in Figure 21 may not be very complex, it is important because it opens many avenues for increasing the amount of data the application can pull in. The list in this example may have only been two entries but if necessary it could have been extended to over ten entries (or many more if required). This would help solve the issue of DBpedia ontology and resource consistency. As DBpedia resources tend to use different properties to represent identical or very similar information, creating an iterative SPARQL query in this way makes it possible to cover all the possible methods for describing data using ontologies, improving the consistency between musical entities represented by the application.

#### 4.6.2 Handling Genres

The musical genres associated with an artist or release were originally going to be obtained exclusively from DBpedia. However, there was inconsistency across DBpedia pages for such resources, especially artists, so genres were not always obtainable. I first noticed this through testing during development when genres were not always obtained by the SPARQL query I had written to do so. Upon investigation of a particular artist which did not appear to have any genre data, this was found to be due to the predicate *dbo:genre* not existing on their DBpedia resource.

As displaying genres was a functional requirement for the system I started to think of other ways of making sure they were displayed. Artists usually had tags associated with them on MusicBrainz so I decided that I would retrieve these for artist information pages, but album pages would still source their genres from DBpedia, as genre data was more often available of DBpedia album resources.

Artist tags on MusicBrainz are assigned by users, and as a result, they are occasionally incorrect or biased. For example, I discovered one artist which was tagged as 'best', which is not objective but someone's opinion, and there are bound to be many more instances of this across the site. Despite this, most of the tags assigned to artists are musical genres, so the tag-lists were useful most of the time. Each tag's 'count' value was used for sorting criteria as more accurate tags tend to be assigned more often. I decided on a limit for the number of tags that could be shown on an artist's page to provide continuity across the application and hide biased or incorrect tags on most occasions, as they are more likely to have a low count value and therefore appear towards the bottom of the sorted tag list. Once the list has been sorted, only the top five tags were passed to artist.html.

Once I had implemented all other functional requirements of the system I returned to the genres section of the artist information pages to see if I could combine data from both DBpedia and MusicBrainz, to provide the user with a more rich and varied knowledge base.

Originally only one line of code was used to assign genres/tags on an artist information page:

```
artist_info['genres'] = get_tags(data['artist'])
```

I then changed the code so that genres were first obtained from DBpedia and assigned to `artist_info['genres']`. After this step the program checked if tags were available from MusicBrainz and then appended them to `artist_info['genres']` one-by-one, provided that each genre did not already exist in the list:

```
artist_info['genres'] = get_dbpedia_genres(artist_info['dbpedia_ID'])
if get_tags(data['artist']) is not None:
    genres = []
    for item in artist_info['genres']:
        genres.append(item[0].lower())

    for item in get_tags(data['artist']):
        if item[0].lower() not in genres:
            artist_info['genres'].append(item)
```

Figure 22 - Code from the `artist_info` function in `views.py`

The local variable `genres` in Figure 22 needed to be created to retrieve the genres already obtained from DBpedia put them in a uniform format so that the tags from MusicBrainz could be compared to them. The `lower()` method was used to format all the strings in the same way so that I could use Python's '`not in`' equivalence operator to ensure that genres were not being duplicated.

This idea of combining both DBpedia and MusicBrainz datasets instead of relying on one for a web page element is an important one because it could be applied to many more functions in the application. Unfortunately, I did not have time to implement this feature in more functions as this implementation was extra functionality that I added late on. In the Section 6.1 Obtaining Data from More Sources of this report, I discuss how I might have approached this task for more of the application's functions.

#### 4.6.3 Creating RDF

As the LinkedBrainz project ended in July 2011, there is a lack of RDF online relating to music artists who have been active since then. In addition to this, RDFa support in MusicBrainz pages has been deprecated and removed as of March 2014, due to its support being 'brittle and poorly maintained for some time' [29]. In a related blog post, what appears to be a MusicBrainz site developer also explained that the RDFa deprecation was approved as they could find 'no one or no application' that 'makes use of the RDFa in our pages'. [30]

In an attempt to rectify the problem of no openly available, up-to-date music-related RDF on the web, I wanted to incorporate a feature into the application that allowed users to download such RDF, in a sense carrying on from where the LinkedBrainz project left off. This way, users could get correct music-related RDF that they can use in creating their own linked data applications if they desired. Furthermore, by providing data in a non-proprietary format, the application would achieve a three-star rating on Tim Berners-Lee's scale for Linked Open Data (see Section 2.2).

Implementation of this extra feature required importing a Python package called RDFlib, which amongst other functionality, allows serialisation of RDF. This meant that a function could be written to create RDF triples related to any artist or album, based on the input parameters of the function. This flexibility meant that accurate and consistent RDF could be produced for different musical entities.

Once RDF is created, RDFlib allows for the use of different RDF serialisations. Therefore, I decided that the user should be provided with RDF in the Turtle serialisation format, based on the following findings from my research:

- 'More human-friendly and readable.'
- 'Not an XML language, designed specifically for RDF.'
  - As Turtle is an RDF specific format based on the Notation3 serialization, three URIs can be grouped to make a triple and information can be abbreviated with the use of triples.
- 'Does not have to represent a graph as a tree so it can be more concise and readable.'
  - XML representations tend to be longer documents with a tree-like structure as both opening and closing tags are required

The above reasons mean that an RDF model can be represented in less lines of code using Turtle instead of standard RDF/XML. Figure 23 obtained from Wikipedia demonstrates this.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://en.wikipedia.org/wiki/Tony_Benn">
    <dc:title>Tony Benn</dc:title>
    <dc:publisher>Wikipedia</dc:publisher>
  </rdf:Description>
</rdf:RDF>
```

```
@prefix dc: <http://purl.org/dc/elements/1.1/>.

<http://en.wikipedia.org/wiki/Tony_Benn>
  dc:title "Tony Benn";
  dc:publisher "Wikipedia".
```

[31]

Figure 23 - Examples of standard RDF/XML (Top) and Turtle serialization (Bottom)

The final solution is a button ‘Export RDF’, which refreshed the web page when clicked to export RDF for the musical entity the user had just been viewing in Turtle. This achieves the three-star rating because the user can then download the Turtle representation by right-clicking on the web page. No password is required, the data is freely available.

The RDF created when the button is clicked aggregates data from the multiple sources used to populate the application’s web pages. Figure 24 shows RDF produced from the artist page shown in Figure 19:

```
<https://musicbrainz.org/artist/9c9f1380-2516-4fc9-a3e6-f9f61941d090> rdfs:label "Muse" ;
  dbp:bandMember "Chris Wolstenholme",
    "Dominic Howard",
    "Matt Bellamy" ;
  rdfs:type mo:MusicArtist ;
  owl:sameAs <http://dbpedia.org/resource/Muse_(band)>,
    <https://www.bbc.co.uk/music/artists/9c9f1380-2516-4fc9-a3e6-f9f61941d090> ;
  foaf:homepage <http://muse.mu/> ;
  foaf:name "Muse" .
```

Figure 24 - An example of a Turtle file produced by the application

Figure 24 shows that data obtained from MusicBrainz such as the URI for the artist’s page and the URL for their homepage is represented in RDF along with a list of band members that were obtained from DBpedia. The prefix ‘mo’ represents the Music Ontology so the information returned is being linked to the Music Ontology, along with the Web Ontology Language (owl) and FOAF vocabularies.

The intended user of the application’s ‘Export RDF’ feature would be web developers who could use the RDF they export to embed RDFa into their own web pages if they contain music-related information or entities such as images or video.

RDFa, an extension to HTML5 for marking up web page content, is used by ‘Search Engines and Web Services to generate better search listings’ (<https://rdfa.info/>). Many sites now utilise RDFa to markup web page elements with reference to schema.org, a site backed by the likes of Google and Bing.

RDFa is used by BBC Music (Figure 25) to link HTML content to schema.org entities:

```
vocab="http://schema.org/" typeof="VideoObject">
<div property="name">Arctic Monkeys talk to Jo Whiley at the BRITs</div>
<div property="image">https://ichef.bbci.co.uk/images/ic/240x135/p01sp9sy.jpg</div>
<div property="thumbnailUrl">https://ichef.bbci.co.uk/images/ic/240x135/p01sp9sy.jpg</div>
<div property="uploadDate">2014-02-21T17:17:00.000Z</div>
<div property="description">Jo catches up with band of the evening Arctic Monkeys backstage at the 2014 BRIT Awards.</div>
<div property="url">https://www.bbc.co.uk/music/audiovideo/popular/p01sp9wy</div>
```

Figure 25 - RDFa embedded in a BBC Music artist web page

Developers exporting RDF from my application would save time as they do not have to write their own RDF. They do not have to research which ontologies music-related RDF commonly uses, as the files exported from the application follow conventions used by the LinkedBrainz project and the Music Ontology.

In addition to the export feature, I wanted to embed RDFa into my own web pages which would take the application to four stars on the Linked Open Data scale, as it would be providing URIs for objects on each page.

```
<div vocab="http://purl.org/ontology/mo/" typeof="MusicArtist" class="main_body">
<h1>Muse</h1>
```

```
<div property="biography" class="description">
<p>Muse are an English rock band from Teignmouth, Devon, formed in 1994. The band consists of Matt Bellamy (lead vocals, guitar, backing vocals, keyboards) and Dominic Howard (drums, percussion).
```

Figure 26 - RDFa embedded within an Artist information page on the application I developed

Figure 26 shows RDFa within the web pages of my own application. The example is taken from an artist information page. The top half of Figure 26 shows the ontology being used to reference data items within the web page, as defined by the ‘vocab’ attribute. Unlike BBC Music which uses schema.org, my application uses the Music Ontology. The ‘typeof’ attribute denotes that all following data marked with ‘property’ attributes belongs to the Music Ontology class ‘MusicArtist’. This class is defined in a div element which contains all data items on the page. The bottom half of Figure 26 shows that the div containing the artist’s description has been given the property attribute ‘biography’. I placed this attribute here as the Music Ontology defines biography as a property of MusicArtist

#### 4.6.4 Getting a track list

Writing a function to obtain all the tracks from a release such as an album has proved very difficult. This is because MusicBrainz sorts these entities into *release-groups* which then contain *releases*. Problems arose because it is only the *releases* themselves that contain an obtainable track listing. For example, an album (one *release group* with a unique MBID) may contain several releases (each with their own MBID) as the album was released on different record labels, in different countries on different formats, with different track lists (for instance, some countries may have received releases with bonus tracks).

The search functionality for albums and the method for getting a list of albums to display on an artist’s home page use only the *release-group* MBID, and it was these IDs being passed to

the *get\_tracklist()* function. This meant another few steps had to be taken from this ID to identify just one *release*, obtain its ID, and then get a track list using this new ID.

An MBAPI function was used to obtain a list of releases, which were returned as dictionaries. My initial plan was for the *get\_tracklist* function to return the UK release as this web application is being made in English alone. The function therefore iterated through the *release-group-list*, using an 'if' statement to check if 'GB' was the value for the key 'Country' within each dictionary item representing a release.

However, through some basic testing of this first implementation, irregularities started to occur. In some cases, no track list was being returned and this turned out to be because there were not 'GB' releases for certain albums. With the ever-increasing popularity of music downloads and streaming via services such as iTunes and Spotify, newer releases were only releases in regions listed as 'XE' and 'XW' on MusicBrainz, which turned out to be Europe and Worldwide respectively. With the Internet accessible worldwide, releases are now often made available in various countries at the same time via streaming services or sites like iTunes.

In the end, the most successful way of obtaining a track list was to find the first release in the list which had a release date which was the same as the '*first-release date*' key item in the original release group. From testing the application with several different albums, more of the albums returned a correct track listing for a normal GB CD release of that album.

The application in its current state still does not handle all track list's correctly though, mainly due to the medium-type of releases. In some cases, only half a track list is returned for an album. After some investigation I found that this was because the release being used was a vinyl format. As vinyl's requiring flipping over to hear the tracks towards the end of LPs, MusicBrainz was returning such releases in multiple dictionaries, with different track lists relevant to different sides of a vinyl. The same issue is true with any multiple disc CD releases.

The final issue encountered when developing this function was speed. From testing it became apparent that it was taking a long time to render album information web pages, sometimes over 20 seconds. This was because the function had to iterate through all releases in a *release-group* to find one that matched the 'first initial release' of the *release group*. I discovered that the function was taking a long time because the for loop would not stop once it found an entry that matched the first release-date. Instead, it was iterating through the whole list to try and find other instances that met this criterion. For historically-significant albums, with many releases on different mediums and several re-releases such as celebrations and remastered editions, the delay this function caused to rendering a webpage was a problem.

I managed to roughly halve the time the function took to execute by refactoring the code and using a break condition in the function's 'if' statement. This meant that if a release matching the necessary criteria was found, the code would break from the 'if' condition and use the dictionary found to extract the track list without iterating through all the other *releases* within the *release group*. Only one *release* was required to collect a track list, so the function could stop once one had been found.

#### 4.6.5 Dealing with ambiguity when getting descriptions

The function used to obtain descriptions underwent several iterations until it worked correctly. It uses a SPARQL query to obtain the *dbo:abstract* property from the DBpedia page for the entity in question.

Problems arose designing this part of the system because there are no strict naming conventions on DBpedia like there are on MusicBrainz with MBIDs. This meant that several different potential URLs needed to be used for each entity until the correct description was found. For example, the album Parachutes by Coldplay on DBpedia could be any of the following URIs:

- <http://dbpedia.org/resource/Parachutes>
- [http://dbpedia.org/resource/Parachutes \(album\)](http://dbpedia.org/resource/Parachutes_(album))
- [http://dbpedia.org/resource/Parachutes \(Coldplay album\)](http://dbpedia.org/resource/Parachutes_(Coldplay_album))

For this album the second URI is the correct one. It is therefore the only URI that can be used to successfully return results in a SPARQL query. However, from looking at several different artists and albums on DBpedia, all three of these naming conventions are used.

To deal with this, the functions for rendering artist and album pages was originally using several 'if' statements and assigning some variable multiple times which was inefficient. The final solution is a list of potential URIs which is iterated through using a 'for' loop. An 'if' statement checks if each URI returns a blank description, and if not, whether the description contains specific phrases are present to ensure that the URI is for the right entity. For example, a description about an album must not only be blank, but also include the word 'album' for it to be the correct URI. Referring to the example URI's above, if the word 'album' was not required the first URI may have been incorrectly used, returning a description about parachutes to slow motion through the air, instead the Coldplay album

Another issue regarding descriptions that occasionally arises when using the program is DBpedia abstracts for very large acts being displayed on the pages of smaller artists if they share the same name. Images of artists are also prone to this problem as they are also obtained using DBpedia URIs (this is not the case with albums as I was able to obtain cover art images from MusicBrainz using release group MBIDs). This happens because a small, relatively unknown artist is very unlikely to have their own Wikipedia (and therefore DBpedia) page. Therefore, if they share the same name as a popular artist, the application tries the popular artist's URI and a description is indeed returned so it displays this on the smaller artist's web page.

#### 4.6.6 Key Errors

Towards the end of the implementation phase, as I tested parts of the application during coding, Python key errors were frequently occurring. Python returns a `KeyError` if a certain key is not found within a dictionary. The MBID function I was using to retrieve data from MusicBrainz was returning a list of dictionaries, based on what data could be found about an artist or album. If any data to be collected by the MBID function call in the code is not present on MusicBrainz, that dictionary key is not returned. This meant that the get methods that I created to extract information from MusicBrainz dictionaries had to be able to handle key errors, otherwise entire web pages would not be rendered. Figure 27 is an extract of code from the system's 'get\_location' function that shows how I used 'try' statements to prevent the errors from occurring.



```

try:
    return data['begin-area']['name'] + ", " + data['country']
except KeyError:
    pass

```

Figure 27 - Code from the `get_location` method in `views.py`

The program attempts to run the `return` statement but if the variable `data`, a dictionary populated with MusicBrainz information, does not have the keys `'begin-area'` or `'country'` then the `except` condition is run. The `pass` statement of this condition indicates that the program should ignore the error and move onto executing the rest of the code. Unfortunately, I did not have time to implement a more elegant solution which would improve the application's performance with regards to usability principles such as visibility and feedback. In Section 5.5 Performance and Error Handling, I will describe a potential way in which the application could be improved to increase the levels of feedback given.

Type errors occurred regularly when implementing the `create_RDF` function in the program. As the RDFLib package I downloaded to implement this feature needs URIs and literals to be explicitly defined with use of RDFLib functions, errors occurred if the strings passed to the function were invalid. This was common for the same reason that key errors were appearing in other parts of the application; some artists did not have certain data items associated with them on MusicBrainz. If data was absent the RDFLib function received no value, hence the Type error.

More error handling was required to handle bad requests being sent to the application's data sources. Exceptions were needed for an `'EndPointInternalError'` which is when a SPARQL endpoint returns a 500 error code meaning it cannot handle a request and `'musicbrainz.ResponseError'`, for bad MBAPI function calls.

#### 4.6.7 Refactoring

The functionality of the application is dependent on many `'get'` methods, many of which originally took an MBID as an input parameter which was then passed to a MBAPI function to retrieve relevant data and store it as a variable. The rest of the function would then extract the relevant data from the dictionaries that are returned. This worked correctly, but from testing it became apparent that running multiple MBAPI functions took a long time. The function used to render the webpage itself (`artist_page()`) featured all of these `get` methods so rendering the page took a long time. In the case of artists with long careers and many releases, it would take even longer to load the page, sometimes over 30 seconds.

To fix this problem I needed to reduce the number of MBAPI function calls, so I refactored the code so that the an MBAPI `get` method was only used once in the functions for rendering the HTML pages. This meant changing the parameters of the MBAPI function so that it pulled in a lot more data, not just facts relevant to individual `'get'` methods, all the data needed to be gathered and stored in one variable. The input parameters for my own `get` methods were then changed so that they instead took the now single data variable as an input. Each of my own `get` methods then had to parse the dataset differently to extract the relevant information for that method. Re-writing the functions in this way made it harder to get them working correctly, but it was worth the effort as the web application began to respond much more quickly as a result.



#### 4.6.8 Implementation on a Web Server

Towards the end of my time on the project I wanted to try and implement the application on a public web server, to prove that this application would be a viable solution on the World Wide Web.

The application is currently hosted at <http://ajh953.pythonanywhere.com/home/>. All features work as expected apart from the Export RDF feature. When the button to generate RDF is clicked on the original version of my application which is local to my PC, a text file entitled either *artist-rdf* or *album-rdf*, which contains the RDF triples relating to the artist or album being viewed by the user, is created on my PC's hard drive. These files are overwritten each time the button is clicked on an information page for a different entity. The user can then view these files in Google Chrome if they have a free web server extension installed that allows access to the RDF files kept on the PC's hard drive.

Exporting RDF therefore does not work on the public website as the RDF files are created in my personal file directory hosted at pythonanywhere, which is private. Given more time for the project I would investigate if there is a way to host a public files folder on pythonanywhere and also change what directory the files are being saved to. This latter of these points would mean more investigation into the capabilities of the RDFlib package.

## 5 System Testing and Evaluation

### 5.1 Testing methodology

The testing method I used could be considered as 'white box' testing because I tested the functions I was working on and made changes to them during development. To do this I was using the 'build' feature in Sublime Text 3 that allows you to run Python code and see its outputs in Sublime, so a separate command line window does not have to be opened. So that I did not accidentally break aspects of the website during development, I created functions in a separate Python file to `views.py`. Only once each function was tested with five different inputs yielding expected results would I then copy the function I had created into `views.py` and then test it on the website itself with the same inputs.

I deemed writing automated unit tests using Python's inbuilt module unnecessary as a lot of the code I wrote used the MBAPI. This is a stable release, which means it would have been tested by its creators. Furthermore, automated testing works well when functions output predictable variables (e.g. a simple function that adds two numbers together). However, as my application retrieves data from several different live sources which can be edited at any time by members of the public, the data outputted by the application could often change. If I was to write automated tests that expect outputs at a certain time based on one of these data sources, tests that originally passed may unexpectedly fail at a later date. Writing code for automated tests would have also been time consuming and I was learning what I needed about the program simply by printing the outputs from functions in Sublime text until they were ready for incorporation to the application.

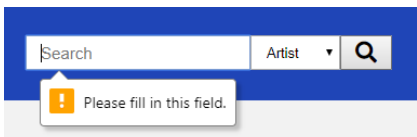
For the above reasons I therefore decided to use test cases instead of automated testing to test specific aspects of the application. I then evaluated the system more generally with respect to three main areas:

- Meeting requirements
- Accessibility and usability
- Performance and error handling

### 5.2 Test Cases

The test cases that I designed evaluate how well the application deals with user queries, navigation of the site and the quality of data returned. The environment used for all the following Test Cases was a laptop machine running Windows 10. Google Chrome was the web browser used to view the application.

Test Case 1			
Success criteria:	Results returned from search even if there are typos in the search query.		
Preconditions:	User is connected to the Internet and on the application homepage.		
Steps			
Step Number	Procedure	Response	Pass/Fail
1.	Enter correct spelling of full artist name into search bar to obtain list of results which further steps can be referenced against.	List of results, including the correct result at the top of the list.	Pass
2.	Enter artist name with capital letters in unexpected places.	Identical list of results to step 1.	Pass
3.	Enter only half of artist's name.	Different lists returned as expected but desired artist is still present in the list when first or last name is used.	Pass
4.	Enter full name with typo in one word	Same list returned as step 3 due to one correctly spelled word.	Pass
5.	Enter full name with one typo in each word.	Different list returned which is much shorter with desired artist absent.	Fail
6.	Enter full name with no spaces between words.	No list returned	Fail
Comments: The search functionality did not pass in every scenario but over half of the searches returned the desired the result which is an acceptable amount. The site could be improved in future so that in instances like step 6 where a mostly blank page is returned is replaced by a message which explains why no results were shown and suggestion that the user tries again.			

Test Case 2			
Success criteria:	If no input is entered the search bar does not lead to an error page (Must work on all application pages.)		
Preconditions:	No characters have been typed into the search bar text area.		
Steps			
Step Number	Procedure	Response	Pass/Fail
1.	Click the search bar 'Enter' button.	Alert box appears when button is clicked asking the user to fill in the search bar. 	Pass
Comments: The current web page remains unaltered and no errors are generated which could confuse the user. Application gives appropriate feedback, precisely indicating what the problem is and asking the user to provide a search query.			

Test Case 3			
Success criteria:	No artists have incorrect, duplicated content on their pages.		
Preconditions:			
Steps			
Step Number	Procedure	Response	Pass/Fail
1.	Search for different artists until the results list returned contains two different artists with the same name.	Home page with search results.	N/A
2.	Open one of the artist names in the list that has appeared more than once in a new browser tab.	User is taken to artist information page.	N/A
3.	Using results list, open information pages for all other artists which have a name identical to the first. Compare information on all open pages.	Artists with identical names have the same description and image.	Fail
Comments: Discovered KeyErrors when trying to open different artist pages. These issues were rectified using Try/Catch blocks in Python. Once these errors were dealt with			

Test Case 4			
Success criteria:	Website displays all data in the same order without any overlapping even if window is resized.		
Preconditions:	User is on an artist or album information page.		
Steps			
Step Number	Procedure	Response	Pass/Fail
1.	Resize window using horizontal drag handles.	Eventually search bar button and drop-down menu can be displace below the text input field and outside of the blue banner it belongs. The site logo also does the same. The description, which should be the first element displayed after the title of the page can get displaced under the side-bar elements.	Fail
2.	Resize the window using vertical drag handles.	Web page elements all remain in the same order.	Pass
Comments: The fact that the application’s element structure does not remain the same when the window is made narrower shows that the application is not ready to be used in a mobile environment. Some additional CSS styling, probably using Bootstrap, would be required in this case.			

### 5.3 Meeting Requirements

#### 5.3.1 Functional Requirements

Before work on the project began requirements for the application were recorded in a document (Appendix A). Upon reviewing the application, it meets 10 of the 11 functional requirements which concern what data items must be returned. There was one additional functional requirement (System database contains at least 50 artists) but it is no longer applicable to the system. It was written before I knew if it would be possible to obtain all data dynamically via SPARQL and an API. If this was not possible, I wanted to download a music related dataset that I could use, so I chose an arbitrary number of artists with associated data which could be used to build early prototypes. However, as the application has access to the MusicBrainz data set, it can obtain data for thousands of artists.

The one functional requirement that was not fulfilled by the application was the ability to retrieve the names of instruments played by band members of artists which are musical groups. This requirement is certainly feasible, it just requires more project time to develop. Essentially, the data could be retrieved by a similar SPARQL query to the one used for obtaining band members, but more lines would be required in the *WHERE* clause to look for the property *dbo:instrument* and then more code to get the label of the instrument, which would be a literal so it can be returned by the query. Adjustments would then have to be made to the Django templates for artist pages to display the instrument name in brackets next to the musician's name or even a mini icon of the instrument they play.

It should be noted that data is not returned for every artist and album available for selection on the home page, but this is due to the availability and consistency of the data sources, so it was expected. I feel that the application still meets the functional requirements as they were more about the capability of the application and the application can pull in all required data so long as it is available.

### 5.3.2 Non-functional requirements

A non-functional requirement for both artist and album web pages was for them to display images, which the application does. Album pages also have track listings whilst artist pages have links to social media sites. One requirement specified that a link to an artist's 'bandsintown' page should be present amongst these links but Songkick was used instead because it provides the same service, but it is a more popular and feature-rich site. Links to setlist.fm and other useful sites such as Genius and Soundcloud were also included.

One of the non-functional requirements was for artist pages to have a selection YouTube music videos displayed at the bottom of the page. Unfortunately, this feature was not implemented. The great thing about the BBC Music site is the number of videos and radio clips it makes available for its users, so I wanted to create something similar which did not offer BBC content alone. YouTube is utilised by most music artists in current times as it is free for its users, so uploaded music videos can attract a huge amount of views. I wanted to see if it was possible to implement a function that runs in the background of the application which searches YouTube using an artist's name and returns a predetermined number of the search results. Implementing this feature would require research into the capabilities of the YouTube API (<https://developers.google.com/youtube/>).

### 5.4 Accessibility and Usability

I feel that the application I have developed is more accessible than MusicBrainz as text is larger and there are fewer elements on each page. Users would also find the application easier to use and learn as they have fewer links and pages to click through than on MusicBrainz, especially to find album data. The search bar on the MusicBrainz home page also loads an entirely new page when a query is submitted whereas my solution just adds the query results to the page the user is currently on.

My application also provides a simpler way to browse Linked Data, such as that found on DBpedia, without needing to understand SPARQL queries. For a user with no knowledge of ontologies, web pages of my application are also easier to read through than DBpedia resource pages as properties are replaced with headings, there is less data on the page and data is not duplicated. I also believe that my application is also more visually appealing.

That said, the application does not look and feel as intuitive and modern as the BBC Music site, but it does offer more discography and biography related data. However, the application does not provide the vast quantities of information that sites like MusicBrainz provide. If someone was looking to make a commercially viable website based on the prototype I have developed during this project which challenges the likes of MusicBrainz, it would need to obtain much more discography data but still represent it in a clearer way than MusicBrainz. For example, this could be achieved by using modern web development technologies such as REACT, Node.js and Angular JS so that vast quantities of artist or album information can be contained on one page but only made visible when the user clicks on certain features.

Figure 28 is an edited version of the artist information page shown throughout the report. It depicts a solution where the discography data shown on artist pages could be extended without overcrowding the page with content through the use of collapsible ‘accordion’ sections.

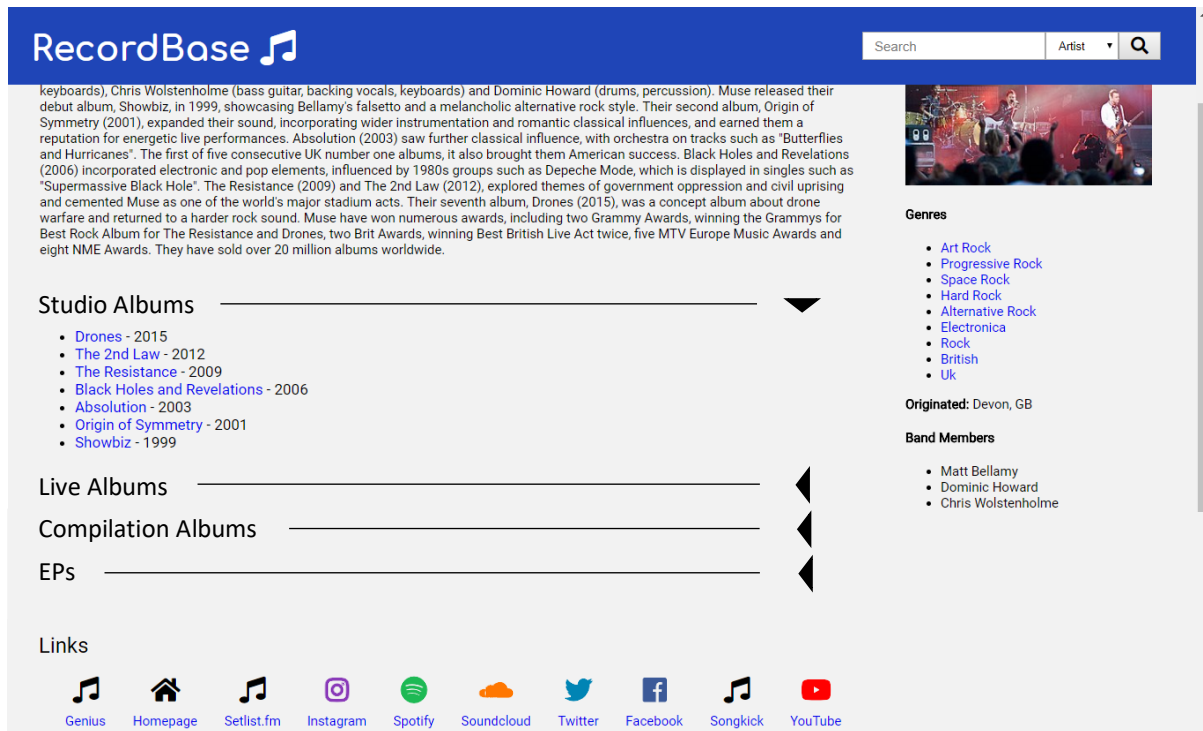


Figure 28 - Artist information page with extended release information

This is a good example because the quantity of releases shown is one of the main aspects of the application that could be improved upon as the application currently only shows albums listed as studio albums on MusicBrainz.

## 5.5 Performance and Error Handling

In terms of speed, the application performs well considering that the data it uses is coming from an external source. The fastest part of the application to render is the search page, which usually returns results in under 2 seconds. Artist pages usually load up in under 10 seconds, typically only just over 5 seconds. Album pages take a similar amount of time.

Assessing performance in terms of accuracy of data returned, the only major issue is with small artists inheriting incorrect DBpedia data regarding world famous acts with the same name (discussed in Section 4.6.5).

With regards to error handling, the application can handle most errors so that Django error pages are not shown to the user. Instead, HTML headings for absent data are also not rendered, so that it is not obvious that content is missing. However, this is not an effective method of rendering the web page if many (or all) data items could not be retrieved. In such cases the user is presented with a mostly blank page with no explanation as to why, which is poor feedback and visibility. To address this issue, I would use a counter in the `artist_page` and `album_page` functions of `views.py`. If an item of data cannot be retrieved by a 'get' method, this counter value would be incremented by one. I would then use an 'if' statement in the HTML templates to check the value of this counter. If the value is above a certain

threshold value for absent data items, a message apologising to the user and explaining why there is such a lack of data will be displayed.

### 5.6 Viability of Implementation Method

Developing a similar web application in this way for a commercial space would be good for reducing overheads related to database acquisition and management because a database is not required. However, such a structure has its disadvantages. Firstly, the owner of the application is not responsible for the acquisition and maintenance of the data it presents, so amendments cannot be made if inconsistencies or flaws are reported. The owner must rely on the administration teams behind the three sites I have used to build my application to maintain high quality data



## 6 Future Work

This section of the report will discuss any work that I could continue to do in this domain given more time. For each idea listed, I will briefly outline the initial steps either myself, or anyone else would need to take to implement the ideas in the application. Some of the topics and ideas were considered towards the beginning of the project and some are new ideas that came to prevalence towards the end of development.

### 6.1 Obtaining Data from More Sources

As it currently stands the application uses data from three different sources but each application element, apart from the genres list on Artist information pages, only uses one data source. For example, a list of producers on an Album page comes exclusively from DBpedia. This was the case due to the limited time I had to complete the project and I wanted to meet all functional requirements regarding what data should be displayed by the application. A few hours into the implementation I decided that I needed to persevere with retrieving data from the most consistent source alone for each element, to make sure they were all finished on time.

A greater variety of data could be displayed on the application's web pages if data from the multiple sources was 'mashed up' before rendering a page. However, if this was the case the application would need to be able to deal with numerous issues, the main two being: recognition of duplicate values across data sources and the calculation of a score for each data item, which could be used to discard incorrect or invaluable data. My implementation of the `artist_page` function to display a mixture of DBpedia genres and MusicBrainz tags as an artist's associated genres deals with the first of these problems when items are explicit duplicates (see Section 4.6.2). However, if more datasets were to be used then the application would have to be capable of recognising duplicates that are less apparent. For example, if multiple datasets are used to obtain genres associated with an album, the genre 'Rock' may be obtained multiple times but in different forms. It could be listed as 'Rock', 'rock', 'Rock Music' so the application would have to recognise this and make sure all these different instances are recorded as only one name (i.e. all three of these tags are listed as 'Rock' on the application which is only displayed once in a web page genre list).

The second of these tasks, assessing the accuracy of different data items so that they can be sorted accordingly, is a much more complex task. In Section 4.6.2 I explained how I attempted to discard biased or incorrect MusicBrainz tags based on their 'count' value but this solution would not work in all instances, especially with datasets like DBpedia where there is no count value to use to the same effect. This issue of assessing data quality was a big part of this project and one that requires far more research to develop an understanding of how to program appropriate solutions.

In addition to 'mashing up' datasets on the application's page, more data sources could also be used in the data collection phase. For example, Genius.com song information pages contain genre tags, so these could potentially be included in the genre lists retrieved for their parent album from DBpedia and MusicBrainz. Genius do not offer Linked Data but they do have an API which can be used to retrieve song, album and artist information. Genius also provide metadata for musical entities so the API may be useful for other application elements of my application, or even entirely new ones. For instance, using the API to get song information via a unique identifier would return that song's lyrics. Successful implementation of feature would mean that track lists on the application's album pages

could be a series of hyperlinks to song information pages that contain song lyrics and metadata.

Genius is just one source that could be utilised; discogs.com, setlist.fm and songkick.com all offer their own APIs that could prove very useful for increasing the volume of data available through the application in the absence of up to date, consistently-reviewed, music-related RDF.

## 6.2 Icons Indicating Data Provenance

With sites like MusicBrainz and DBpedia it can be difficult to identify the authors of the data they host. As the application I have developed uses data from multiple sources it would be good to provide transparency regarding data provenance. This could be achieved by adding some form of indicator next to data on the application which shows which data source it came from. I feel a small icon would be better for this to make it as subtle as possible, so that it is there for users who are interested but it does not detract from the rest of the content on the page.

Figure 29 is a screenshot of the side bar from an artist information page edited to show two MusicBrainz logos and one DBpedia logo to demonstrate how this feature might look once implemented.

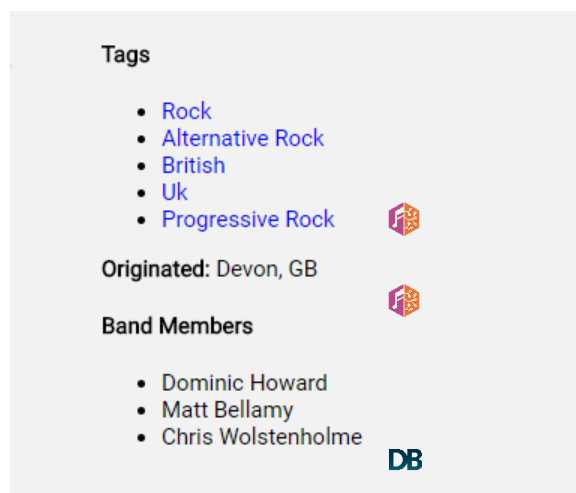


Figure 29 - Illustration of how data provenance icons would be displayed

I do not feel this feature is essential in the applications current state as each web page element only gets data from one source. However, if more data sources were to be utilised as I described in the previous Future Work section, then this feature would be more important as it will become harder for users to track the origin of the data. In cases where lists are formulated using multiple sources, multiple icons would have to be displayed next to that application element. To add to these icons' functionality, they could even be contextual hyperlinks to the data source itself, e.g. when a 'DB' icon is clicked the user is taken to the DBpedia page for the artist they are currently viewing.

## 6.3 Adding a Database to the Application

The application I have developed does not have its own database; it relies on the MusicBrainz, DBpedia and BBC Music data being available to run correctly. Part of the challenge and intrigue of this project was investigating if a web application like the one I have developed can offer users accurate information and run at an acceptable speed with

minimal errors. That said, adding a database to the application would open up new possibilities which could both enhance existing features of the application and allow for the creation of new ones.

The first type of database that could be added would be used to store RDF data. Such databases are known as RDF triple stores. The RDFlib package currently being utilised by the application allows all data represented by the application to be converted into RDF, even if data was obtained using the MBAPI which means it was not originally in an RDF format. The use of this package would have to be extended from simply providing the user with the ability to download RDF files for a triple store to be possible. A function could be written in the application which iterates through artist and album information dictionaries and uses the RDFlib package to parse this data into RDF variables which are then stored in the RDF database. The Sesame framework discussed in Section 2.5 Tools Used to Build Application would be a logical choice for implementing such a database.

Implementing an RDF database in the application would have a couple of benefits. Firstly, if the database is written to every time a user visits a page, the load time of these pages could be reduced in future if the application uses its own RDF data to populate the page instead of running MBAPI function calls and SPARQL queries again. If a program could be written to automatically run the functions necessary for gathering data for thousands of artists and music releases, the whole website could then rely on its own RDF database as a data source instead of always relying on external sources. These external sources could then only be called upon as a backup. Making the application run this way would make it more fault tolerant as it can still be functional even if accessibility to MusicBrainz or DBpedia becomes restricted or impossible for periods of time due to technical issues or maintenance. However, such a function would take a while to implement and it would also have to be efficient as it would need to be run frequently to ensure the database is kept updated. Furthermore, if the application had many more users and it still relied on gathering the data in the moment, SPARQL queries to DBpedia could be executed concurrently which would slow down the application's response times. Access times would be faster and much more predictable if the application was supported by its own database.

Secondly there would be an opportunity to increase the capabilities of the application's search functionality. Currently the application relies on two MBAPI search functions to return results based on the user's queries. It is necessary for the MBAPI function to be executed with each new search as the application's content is generated dynamically. Using SPARQL queries to the application's own database would allow a much greater number of options to be added to the search bar's drop-down menu and also allow for complex filtering of datasets as SPARQL queries offer a huge amount of flexibility in queries, in exactly the same way as SQL does.

#### 6.4 Wider Search Capability

The web application developed utilised known-item searching, meaning the user needed to at least know the name of the subject they were searching for additional information about (i.e. artist name or album name). However, the site would be more useful if it could also facilitate subject searching. Subject searching in this context would be where the user does not intend to search for a piece of music but may instead want to browse all work that falls within a certain musical genre.

Libraries tend to reflect a model where album titles are used as the primary method for cataloguing, but it is documented that people are now searching for the titles of individual tracks on music libraries. This is likely due to the omnipresence of music which has become possible with modern technology and media, coupled with today's 'on demand' culture.

People hear songs everywhere from adverts and the radio to television and movies. When people hear individual songs they like, they are going to want to seek out those tracks specifically by title on services such as Spotify or Apple Music, should they know the track title. It would therefore be beneficial to users if the search functionality of the application was increased even further to allow searching for specific songs.

These additional features would have been possible to implement with the same MBAPI search function that I used to create a search bar for just artists and albums. There were function parameters that could be changed to obtain artist and release tags via search as well as releases themselves.

The ideal search capability of the website would allow users to search for all entities on the application such as artist origin locations, producers, release years and track names. For this to be possible the data on the site would have to be linked in the same way that data on DBpedia is. This would require a database to hold all artist and album information and the current solution does not have a database because all the site's content is dynamically generated, and in a sense, using DBpedia, BBC Music and MusicBrainz databases like its own.

### 6.5 More links and DBpedia content

With more time for implementation, DBpedia could have been used to provide descriptions of band members. Individual members of very famous bands are often themselves very famous, so they often have their own DBpedia pages. This could have been utilised so that such band members on artist pages are hyperlinks to pages about the individual, which offer an even wider set of information for consumption by users. Birth date, birth place, associated bands and images are just a few items of information that could have been obtained using SPARQL queries.

Artist origin locations could have also been hyperlinked to information pages and/or maps in much the same way.

### 6.6 Identification of missing content

Whenever web pages are generated in the application, any headings related to information that could not be obtained from the data sources are not shown. This was to prevent confusion for the user, as headings with no content below them would be confusing.

Missing content was not logged but it would be a great addition to the application if it was. A log of missing content could be used to identify gaps in DBpedia's knowledge base, and it could be used as a source for contributing to DBpedia. Once the missing information has been added to DBpedia, it would be automatically obtained by the SPARQL queries used by the application, hence the application would also be updated in turn.

Over time these efforts would not only make the application a more consistent and informative resource for the user, but others utilising Linked Open Data will also benefit. As already discussed, the idea of the Semantic Web is useful for AI systems and with their

increased use and popularity, adding to the LOD cloud would only allow such systems to work more effectively, in turn benefiting users.

## 7 Conclusions

### 7.1 Reviewing Aims and Objectives

#### 7.1.1 Research

*Aim:* Study different approaches available for building a linked data driven web application and choose a suitable platform for implementation.

When beginning work on this project, the plan was to use Linked Data sets as sources for the entirety of the application's content. In this scenario my first task would have been to decide whether the system I developed should have its own RDF triple store, or whether it would be feasible to retrieve all data dynamically using SPARQL queries. Therefore, my research would have been more focused on what semantic web programming methods would have been the best to use based on my chosen system architecture. This would have involved research into whether it was best to use Python or Java, as there is a well-documented Java framework called Jena which can be used for Semantic Web programming tasks. I would have needed to decide whether it would have been better to use more traditional methods using Jena or attempt to use newer methods in Python. Furthermore, if I had decided on retrieving content periodically and storing it in my own RDF triple store, research into methods of achieving this would have been required.

However, as I quickly realised that there was a shortage of music-related Linked Data sets that were up to date or still available, my research became focused more on how I could retrieve up-to-date information. Discovery of the Python API that gave me access to MusicBrainz data allowed me to do this, so that then made it easy for me to decide on using Python instead of Java, so less research into the capabilities of these two languages was required.

Because of these findings, the rest of my general research into the Semantic Web was focused on how I could exercise good practice on my own application with regards to publishing Linked Data, in an attempt to provide freely accessible music related RDF.

#### 7.1.2 Identifying Sources

*Aim:* Identify sources of data and build a data collection system to query multiple data sources, integrate and pre-process the results for publishing.

I successfully identified three sources of data for the application. The data collected was also pre-processed. For example, Python dictionaries of data obtained from MusicBrainz were not simply printed out, only key values of information were selected from them. In the case genres returned from DBpedia, formatting was applied so that the text was treated as a title and the first letter of each genre was capitalised. This way the formatting between each genre was consistent. Integration of data was achieved across whole artist and information pages, as the data on such pages came from two or three of the sources. At the level of an individual function, data was only integrated when genres were collected for artist information pages as both MusicBrainz and DBpedia was used to source this section. To improve the application, data could be integrated in other sections of each page to provide the user with more information.

### 7.1.3 Designing an Application

*Aim:* Design and implement an interface component of the application which allows the filtering and extraction of specific datasets and the presentation of the data in an appropriate format.

A user interface for the application has been design using Django. The only filtering that the user can do is with the search bar which includes a drop-down menu for selecting whether a list of artists or albums should be returned when the user's query is submitted. The ability to execute more complex queries of Linked Data and MusicBrainz data could have been achieved if the search bar on the site was served by a function that could support the construction of more complex SPARQL queries, instead of solely relying on a MBAPI function.

### 7.1.4 Testing Application

*Aim:* Test and evaluate the system with multiple datasets on the linked data cloud. Produce web pages that are automatically generated based on the content the user is looking for.

I was unable to test the application with multiple datasets on the LOD cloud as each function in the application is coded to query predetermined datasets. For example, the function *'get\_band\_members'* is built to query DBpedia alone as it was the best data source for retrieving such information. The function would not work correctly if passed the URI for a different Linked Data set.

I decided to build the application this way due to the state of music-related data in the LOD. As discussed in Section 2.4.1 MusicBrainz, MusicBrainz RDF data is seven years out-of-date and I wanted to build an application that could provide users with up-to-date information. Unfortunately, there were not any other music-related related datasets in the LOD cloud suitable for use.

I feel that testing the functionality of the application itself is a big area for improvement in the project. It would have been better if I started the testing phase earlier which would have given me time to produce more test cases. I do not feel that I did enough test cases and think it would have been better if I produced around 10-12 of them, which would have covered more aspects of the application's functionality.

## 7.2 Achievements

Despite varied accomplishment with regards to the original aims and objectives of this project, I have overcome numerous unexpected problems during implementation and created a functional and useful application. Noteworthy achievements include:

- With the absence of up-to-date music related RDF on the LOD cloud, the applications 'Export Data' functionality allows users to obtain RDF which is not only up-to-date but conforming to common specifications in the field such as RDF Schema, Friend Of A Friend and the Music Ontology. The Web Ontology Language property *'sameAs'* has also been used to link the MusicBrainz URIs of entities to their respective DBpedia and BBC Music URIs. Providing such data in a non-proprietary format is equivalent to a three-star score on the Linked Open Data rating scale (see Section 2.2). This was extra work as this feature of the application was neither a functional or non-functional requirement in the original specification document.

- RDFa has been embedded in artist and album information pages. An extension for HTML web pages, RDFa can be used by search engines to enhance results as it explicitly defines data items and the relationships between them (see Section 4.6.3). An example of a way such results could be enhanced is context-specific data provided alongside the data the user originally wanted to retrieve with their query. If the application was developed to a level of commercial viability and its pages were permanently available (instead procedurally generated each time the application is used) this RDFa could be effectively used by search engines such as Google. Another application function that was not an original requirement, including RDFa like this achieves a four-star rating on the LOD rating scale.
- Originally the 'Genres' section on artist pages only retrieved tags from MusicBrainz, which are assigned by users and can occasionally be biased or incorrect as a result. In the last couple of weeks before the project deadline, I altered the code so that this section of the web page now incorporates an artist's DBpedia data too. If a DBpedia genre is the same as a MusicBrainz tag, it is not added to the list. By combining these two datasets the issue with bias is negated as DBpedia genres are well established and cannot be subjective or invented by its users. The user also gets a bigger picture as more genres are returned.
- SPARQL queries to obtain data from DBpedia originally used only one property. This meant that no data was returned when a resource did not have that property, but the required information may have been assigned to a slightly different property. I amended the application so that some of these functions now iterate through a list of potential properties, running the SPARQL query with each property to increase the likelihood of data being found and rendered on the application's web pages.
- Incorrect descriptions are returned from DBpedia less frequently than in initial iterations of the application as the function responsible for this feature has been altered to check if certain conditions are met before returning the description.
- Extra functionality was added to show a link to a BBC Music review of an album if there is one available.



## 8 Reflection on learning

To create my application, I had to learn how to use Python's Django web framework during the implementation phase as I had never used it before. Despite this being time consuming I feel it was worthwhile as I produced a more user-friendly website and feature-rich application than I could have done using the School's projects server. By not using the School's web server I also gained experience in deploying a Django project on a third-party server. I feel that experience in these areas has made me more employable to companies looking for web developers; the whole implementation phase of the project has been great for my personal development as a programmer.

I have also learned a lot about the uses and limitations of the Linked Open Data cloud, especially DBpedia. By learning how to retrieve data from this cloud using SPARQL, I can extend what I have done with music-related RDF to any future projects I may encounter during employment. For instance, in a project involving the extension of an online dataset's scope, I would be able to suggest the use of LOD cloud as a way of obtaining more context-specific data without having to request access to datasets belonging to a third-party. In a commercial setting this could save a team time, and potentially money, as they would not have to negotiate with another company for the acquisition or sharing of data.

With regards to the Semantic Web and Linked Data, I think my work in this project has highlighted the need for a review of the state of music-related Linked Data and whether it is worthwhile updating. The fact that MusicBrainz took the decision to move away from embedding RDFa in their web pages and contributing RDF data dumps to the LOD cloud indicates that they do not feel it is a worthwhile pursuit. I do not think that they disagree with the benefits that the concept of the Semantic Web offers. They still offer vast amounts of music data that can be accessed with the use of an API and their Next Generation Schema initiative. However, JSON is used instead of RDF to embed such data in their web pages and it is these JSON representations that the MBAPI uses to retrieve data.

The idea of the Semantic Web is very much alive with the AI technology that is available in current times. Google searches are more sophisticated providing users with results which are complimented with additional content specific information and voice-operated devices such as Amazon Alexa and Google Home are very popular. Such AI technologies rely on effective knowledge representations of which ontologies are hugely important. However, the transition to JSON made by MusicBrainz may indicate an industry wide shift in the technologies used to deliver the Semantic Web.

For instance, Figure 25 in Section 4.6.3 shows RDFa embedded in a BBC Music page using the schema.org ontology but they now also embed Open Graph Protocol properties in their artist pages. Inspired by microformats such as RDFa, the Open Graph Protocol was developed at Facebook and it allows web page elements to be represented within the social graph, which is a network of people on the Internet. [32]

```
<meta name="twitter:site" content="@BBCMusic" />
<meta name="twitter:image:src" content="https://ichef.bbci.co.uk/images/ic/960x540/p02swhx2.jpg" />
<meta property="og:title" content="Lorde - New Songs, Playlists & Latest News - BBC Music" />
<meta property="og:type" content="website" />
<meta property="og:url" content="https://www.bbc.co.uk/music/artists/8e494408-8620-4c6a-82c2-c2ca4a1e4f12" />
<meta property="og:image" content="https://ichef.bbci.co.uk/images/ic/960x540/p02swhx2.jpg" />
<meta property="og:description" content="The BBC artist page for Lorde. Find the best clips, watch programmes, catch up on the news, and read t
<meta property="og:site_name" content="BBC" /> <meta name="viewport" content="width=320, initial-scale=1, maximum-scale=1, user-scalable=0" />
```

Figure 30 - Open Knowledge Graph Properties embedded in a BBC Music artist information page

With similar syntax to RDFa, the '*og*' prefixes in the meta tags of Figure 30 indicate the use of the Open Knowledge Graph properties.

From this evidence and further research in this project I believe that the future of Semantic Web technologies lies with embedded HTML data as practiced by the BBC. The data the MBAPI can retrieve is embedded in MusicBrainz pages but it is represented in JSON not RDF. So MusicBrainz are embedding data in their pages like the BBC but they are just using a different format. Furthermore, I struggled to find RDF data dumps or online resources that are up-to-date and still being maintained, DBTune being the prime example for this specifically music-based application. This again illustrates a shift towards embedded data.

In terms of my application, I believe it would be possible to improve its accuracy with regards to fetching data so that it provides a comprehensive music information service which uses freely available data on DBpedia, MusicBrainz and BBC Music. However, this would require a lot more implementation time to enhance the application's ability to recognise erroneous and duplicated data. Furthermore, if the application was to be commercially viable it would have to be optimised so that it can dynamically retrieve a lot more data without increasing the time taken to do so. Despite this, I am happy with what the application offers given the time I had to create it and the absence of up to date MusicBrainz RDF data.

## 9 Bibliography

- [1] BBC, "Frequently Asked Questions," 2018. [Online]. Available: <https://www.bbc.co.uk/music/faqs>. [Accessed 15 February 2018].
- [2] R. Guns, "Tracing the Origins of the Semantic Web," *Journal of the American Society for Information Science and Technology*, vol. 64, no. 10, pp. 2173-2181, 2013.
- [3] J. Hebler, *Semantic web programming*, Indianapolis, IN : Wiley, 2009.
- [4] K. F. Gracy, M. L. Zeng and L. Skirvin, "Exploring methods to improve access to Music resources by aligning library Data with Linked Data: A report of methodologies and preliminary findings," *Journal of the American Society for Information Science and Technology*, vol. 64, no. 10, pp. 2078-2099, 2013.
- [5] W3C, "Linked Data," 2015. [Online]. Available: <https://www.w3.org/standards/semanticweb/data.html>. [Accessed 19 March 2018].
- [6] "State of the LOD Cloud," 2014. [Online]. Available: [http://lod-cloud.net/state/state\\_2014/](http://lod-cloud.net/state/state_2014/). [Accessed 26 March 2018].
- [7] G. G. Valdez, "Linked Data Overview and Usage in Social," Technical University of Berlin, Berlin.
- [8] T. Berners-Lee, "Linked Data," 2006. [Online]. Available: <https://www.w3.org/DesignIssues/LinkedData.html>. [Accessed 19 March 2018].
- [9] "SPARQL," 2018. [Online]. Available: <https://en.wikipedia.org/wiki/SPARQL>. [Accessed 19 February 2018].
- [10] MusicBrainz, "Database Statistics," 2018. [Online]. Available: <https://musicbrainz.org/statistics>. [Accessed 23 March 2018].
- [11] MusicBrainz, "MusicBrainz Database - Schema," [Online]. Available: [https://musicbrainz.org/doc/MusicBrainz\\_Database/Schema](https://musicbrainz.org/doc/MusicBrainz_Database/Schema). [Accessed 18 March 2018].
- [12] MusicBrainz, "LinkedBrainz," 2015. [Online]. Available: <https://wiki.musicbrainz.org/LinkedBrainz>. [Accessed 19 February 2018].
- [13] MusicBrainz, "LinkedBrainz/RDF," 2015. [Online]. Available: <https://wiki.musicbrainz.org/LinkedBrainz/RDF>. [Accessed 14 February 2018].
- [14] DBpedia, "About," 2017. [Online]. Available: <http://wiki.dbpedia.org/about>. [Accessed 27 March 2018].
- [15] K. U. Idehen, "What is DBpedia, and why is it important?," 2016. [Online]. Available: <https://medium.com/openlink-software-blog/what-is-dbpeda-and-why-is-it-important-d306b5324f90>. [Accessed 14 April 2018].

- [16] DBpedia, "Ontology," 2017. [Online]. Available: <http://wiki.dbpedia.org/services-resources/ontology>. [Accessed 26 April 2018].
- [17] DBpedia, "DBpedia-Live," 2017. [Online]. Available: <http://wiki.dbpedia.org/online-access/DBpediaLive>. [Accessed 28 March 2018].
- [18] Q. Bu, E. Simperl, S. Zerr, Y. Li and M. Sabou, "Using microtasks to crowdsource DBpedia entity classification: A study in workflow design," *Semantic Web*, pp. 1-18, 2017.
- [19] The Music Ontology, "Frequently Asked Questions," [Online]. Available: <http://musicontology.com/docs/faq.html>. [Accessed 16 February 2018].
- [20] MusicBrainz, "LinkedBrainz/NGS to RDF mappings," 2015. [Online]. Available: [https://wiki.musicbrainz.org/LinkedBrainz/NGS\\_to\\_RDF\\_mappings](https://wiki.musicbrainz.org/LinkedBrainz/NGS_to_RDF_mappings). [Accessed 20 February 2018].
- [21] J. Rayfield, "BBC World Cup 2010 dynamic semantic publishing," 2010. [Online]. Available: [http://www.bbc.co.uk/blogs/bbcinternet/2010/07/bbc\\_world\\_cup\\_2010\\_dynamic\\_sem.html](http://www.bbc.co.uk/blogs/bbcinternet/2010/07/bbc_world_cup_2010_dynamic_sem.html). [Accessed 27 April 2018].
- [22] Ontotext, "BBC: Dynamic Semantic Publishing FIFA World Cup Web Site," 2018. [Online]. Available: <https://ontotext.com/company/customers/bbc-dynamic-semantic-publishing/>. [Accessed 27 April 2018].
- [23] DBpedia, "DBpedia Lookup," 2017. [Online]. Available: <http://wiki.dbpedia.org/projects/dbpedia-lookup>. [Accessed 29 April 2018].
- [24] K. F. Gracy, M. L. Zeng and L. Skirvin, "Exploring methods to improve access to Music resources by aligning library Data with Linked Data: A report of methodologies and preliminary findings," *Journal of the American Society for Information Science and Technology*.
- [25] D. M. King, "Catalog User Search Strategies in Finding Music Materials," *Music Reference Services Quarterly*, vol. 9, no. 4, pp. 1-24, 2007.
- [26] Genius, "About Genius," 2018. [Online]. Available: <https://genius.com/Genius-about-genius-annotated>. [Accessed 21 April 2018].
- [27] "Most popular social networks worldwide as of April 2018, ranked by number of active users (in millions)," 2018. [Online]. Available: <https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/>. [Accessed 28 April 2018].
- [28] OpenLink Software, "Tutorial III: Examples based on BBC Linked Data Spaces," [Online]. Available: [https://virtuoso.openlinksw.com/tutorials/sparql/SPARQL\\_Tutorials\\_Part\\_3/SPARQL\\_Tutorials\\_Part\\_3.html](https://virtuoso.openlinksw.com/tutorials/sparql/SPARQL_Tutorials_Part_3/SPARQL_Tutorials_Part_3.html). [Accessed 18 March 2018].

- [29] MusicBrainz, “LinkedBrainz / RDFa,” [Online]. Available: <https://musicbrainz.org/doc/LinkedBrainz/RDFA>. [Accessed 19 April 2018].
- [30] “Our RDFa dilemma,” 2013. [Online]. Available: <https://blog.musicbrainz.org/2013/09/03/our-rdfa-dilemma/>. [Accessed 19 April 2018].
- [31] Wikipedia, “Notation3,” 2018. [Online]. Available: <https://en.wikipedia.org/wiki/Notation3>. [Accessed 27 April 2018].
- [32] “The Open Graph Protocol,” 2017. [Online]. Available: <http://ogp.me/>. [Accessed 27 April 2018].
- [33] N. F. Noy and D. L. McGuinness, “Ontology Development 101: A Guide to Creating Your First Ontology,” [Online]. Available: [https://protege.stanford.edu/publications/ontology\\_development/ontology101-noy-mcguinness.html](https://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html). [Accessed 26 April 2018].
- [34] RDFLib Team, “rdflib 4.2.2,” 2013. [Online]. Available: <https://rdflib.readthedocs.io/en/stable/>. [Accessed 21 March 2018].
- [35] T. Heath, “Linked Data,” [Online]. Available: <http://linkeddata.org/>. [Accessed 26 February 2018].
- [36] J. Nielsen, “10 Usability Heuristics for User Interface Design,” 1995. [Online]. Available: <https://www.nngroup.com/articles/ten-usability-heuristics/>. [Accessed 29 March 2018].
- [37] BBC, “BBC Music,” 2018. [Online]. Available: <https://www.bbc.co.uk/music>. [Accessed 15 February 2018].
- [38] MusicBrainz, “About - MusicBrainz,” 2018. [Online]. Available: <https://musicbrainz.org/doc/About>. [Accessed 15 February 2018].
- [39] Django Software Foundation, “Django Documentation,” 2018. [Online]. Available: <https://docs.djangoproject.com/en/2.0/>. [Accessed 26 February 2018].