

Project 5: Building an Image Database with searchable metadata

Shreya Kapoor, 3200588

Sophia Krix, 3185606

Gemma van der Voort, 3210196

February 7, 2020

LSI b-it Rheinische Friedrich-Wilhelms-Universität Bonn
Programming Lab II
WS 2019/2020
Instructors: Dr. Sebastian Schaaf, Dr. Jens Dörpinghaus

Abstract

This thesis describes the process of building an Image Database with searchable metadata. Searching for an image greatly benefits from having useful metadata to accompany it. The proposed solution allows the user to save the picture together with its online location (url) and any metadata. The application then saves both the picture and its metadata in a SQLite database. This database can be queried and is accessible via a RESTful API service.

Contents

1	Introduction	3
2	User understanding	3
2.1	Determine User Objectives	3
2.2	Situation Assessment	4
2.3	Application Goal	4
2.4	Project Plan	4
2.4.1	SQLite Database Engine	5
2.4.2	Documentation about Tables and Data Structures	7
3	Data Understanding	7
3.1	Data exploration	8
3.2	Project Plan Extension	8
3.2.1	Mapping Image Metadata to Files	8
3.2.2	Creating a Database from Metadata and Image Files	9
3.2.3	Collaboration	9
4	Tool building	11
4.1	Setup of SQLite and Working Environment on Linux	11
5	Implementation	13
5.1	Task01: Mapping Image Metadata to Files	13
5.2	Task02: Interaction with the SQLite Database	14
5.3	Task03: Web Service	18
5.3.1	Background	18
6	Evaluation	23
6.1	Result Evaluation	23
6.2	Process Review	23
6.3	Future Steps	23
6.3.1	Image and Metadata processing	24
6.3.2	Application Access	25
7	Deployment	26
A	Approach: Adapted CRISP-DM	27
B	Statement of Authorship	30

1 Introduction

This thesis describes the design and implementation of a Java-based application to store image files and metadata associated with them.

Image analysis is a highly relevant topic for almost all research fields. Prior to data mining, data curation, and the development of machine learning models, images need to be retrieved, annotated and stored. To develop efficient analysis workflows, the storage and annotation of images is crucial. The application described in this thesis describes an approach to facilitate the storage and retrieval of images and their associated metadata. The goal of this project is to save images and metadata extracted from literature in a database and to handle queries from a superuser and his/her collaborators.

2 User understanding

This section describes our interpretation of what our applications' users will be like. It describes what their needs are and thus what an application should strive to be like. It will conclude with a project plan, where we will outline what capabilities our application should have and why.

2.1 Determine User Objectives

We target our application to users in the scientific world, who will extract images from literature that is of interest to them. They can then possibly go on to use these images for data mining approaches, or simply to have their own tool for indexing and retrieving images that are relevant to their research. For this, they need to be able to create their own database or databases, add images and associated metadata to the database. In addition, working in batch mode could be highly useful, as it allows the user to add whole directories to the database at once. This is useful in the use case that the user did not start out having our application, but only got it later. In this case, the user could have already started gathering images themselves in a folder. It could also be useful when the user downloads multiple images from a single source or just to save time.

In both scenarios, the users will probably not be working alone, but will be collaborating with other researchers who also want to add images and metadata to the database.

In short, the user objectives are:

- create a database
- store images
- store metadata in relation to the images
- store folders of images and metadata simultaneously
- collaborate with other researchers

2.2 Situation Assessment

In this situation assessment, we will describe the constraints and requirements of our application.

1. The programming language used in development has to be Java
2. The time period to develop the application is from November 13th, 2019 to January 15th, 2020.
3. The application will be developed by a three person team consisting of the authors of this report.
4. A folder of images and metadata file to use for testing during development.
5. The relational database management system SQLite should be used.
6. The tool Spring, for application development in Java, should be used.

2.3 Application Goal

In the introduction (1), we defined the application goal generally:

The goal of this project is to save images and metadata extracted from literature in a database, as summarised by figure 1.

Now that we have analysed what our users need and what our constraints are, we can define a more detailed goal:

The goal of this project is to develop an application, written in the Java programming language by the authors of this thesis in the given timeframe, that can create databases, store (folders of) images and metadata files, and allows for collaboration with other researchers.

This goal is summarised in figure 1.

2.4 Project Plan

Based on our application goal, we propose the following application design plan which is illustrated in Figure 1:

The user can upload images and corresponding metadata files. If these metadata files do not yet contain enough information to create a database, the user can give additional meta information that will be added to the existing file. In case there is no such metadata file existing, a new metadata file can be created. When the information about the images is sufficient, the superuser can create a SQLite database and transfer the information in the metadata files as well as the images into the SQLite database. From here on, the superuser can also add further samples into the database and perform queries to extract information.

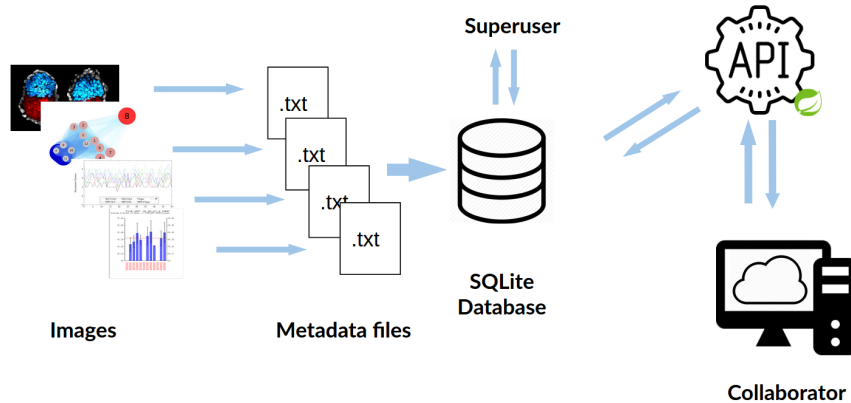


Figure 1: A summary of the problem statement. Scientific literature, stored in databases such as PMC, contain many images. These images need to be extracted

In order to allow collaboration with other users, an RESTful API Service will be connected to the SQLite database. By using this interface, collaborators can store files and query the database easily via a webbrowser.

The application will be designed for two types of users: the database owner and the collaborator. The database owner creates the database, and has full access to all the options via a command line interface. This user should be able to download a .jar file and run the program from command line. The tool SQLite will be used as a relational database management system.

2.4.1 SQLite Database Engine

Using SQLite database as an application file format is advantageous in many ways for the user. It is more than a pile-of-files database with a simple key/value structure. Additionally, an abundance of diverse tables containing different data types can all be linked to each other, can be indexed to allow fast retrieval and can be stored as a compact single disk file ([Hipp,]).

SQLite has a simplified application development. By linking against the SQLite library all requirements are satisfied to enable reading or writing the application file.

A main advantage of the SQLite database engine is its speed. SQLite is faster in reading and writing than a pile-of-files format. The reason for this is that exactly the information that is asked for can be extracted instead of parsing the entire document. This reduces the amount of data drastically that is saved into memory and allows for a much better performance.

As SQLite has few dependencies, it is "self-contained". No external libraries or interfaces are needed for SQLite to work. The additional advantage through

this is that it runs on any operating system. It can be easily downloaded as a single source code file and it does not require any tools to build.

As SQLite is full-featured, its implementation includes tables, multi-column indexes, triggers and views. An SQLite database can be saved in a single file document with arbitrary extension and no naming restrictions. This allows for clear and easy use of the database as a file.

The high-level query language that is used allows for easy retrieval of content that is asked for. By using common command-line tools of Linux the content of SQLite database files can be easily accessed. There also exists a detailed documentation for the SQLite format which is very specific in its definitions. Additionally, the compatibility of SQLite database files is preserved to be able to access database files in the long-term.

SQLite is a cross-platform database engine whose files are portable between 32-bit and 64-bit systems. It is also possible to copy a database between big-endian and little-endian architectures.

As a transactional database, SQLite also follows the ACID (Atomic, Consistent, Isolated, Durable) paradigm. This guarantees that all queries and changes are valid by fulfilling the properties of being atomic, consistent, isolated and durable. Even in the event of errors or crashes of the operating system, this paradigm holds true.

In case of any system errors there is a highly reliable report system that will handle any upcoming error. I/O errors and out-of-memory errors are reported to the user so that he can clearly see what the source of the error is and handle it accordingly. If the input by the user is corrupted, as wrong formats of database files, these errors will be reported, too.

During the progress of an application, rows and column or also entirely new tables can easily be added to the SQLite database file. This is especially important in our case as the user is thereby able to add new entries to the database. These entries can then consist of an image in byte format and the additional meta information in the corresponding columns. One must just take into account that the code concerning these additional columns must be changed accordingly. This makes the SQLite file format easily extensible and very convenient for our use case.

The difference of SQLite to other database engines is that the majority of SQL database engines need a server for their communication. These other programs access the database via a Transmission Control Protocol/Internet Protocol. When using SQLite, the accessing the database does not require a server as an mediator. Because of the serverless behaviour of the SQLite database engine multi-user capabilities are restricted here.

In order to install and configure SQLite there is no separate server process required. Therefore, programs do not need support to set up SQLite but solely require access to the disk. Another characteristic of the "zero-configuration" database engine is that it can be used by more than one application to access the same database simultaneously.

As our application is written in Java, it is a requirement for us that the query language would be able to interact with this syntax. Therefore, it is a

great advantage that there exists an interface for Java of SQLite. The entire documentation as well as the code is open to the public. This displays open-source character of SQLite. The high quality of the code is ensured by SQLite not being open-contribution.

A restriction of SQLite is that the database size is limited to 140 terabytes. For the application of storing huge amounts of data, this could pose a problem. Especially as we store images in a byte format into the database, the size used in memory would increase very fast. Therefore the usage of SQLite restricts us to a certain limit of database file size which will have to be kept in mind.

2.4.2 Documentation about Tables and Data Structures

The user can input images and metadata files that are to be stored. Images of the format .png, .jpeg and .jpg are allowed. The metadata files in the format of .txt files have to follow a specific structure in order to be parsed:

```
Author: author_name
Title: title_name
https://www.link-name.com
```

The SQLite database is the main frame with which the user interacts to store and get data. The SQLite database has a name in order to be uniquely identifiable. Furthermore, it has a path on which the database is saved on the local system of the user as a file.

The database consists of a table named *IMAGES* that stores the information about one sample in a row. The columns contained in the table are listed in the following. They are of the specified type and contain the described information.

AUTHOR: String, the author name of the image or the related article
TITLE: String, the title of the image or the related article
LINK: String, the weblink related to the image
PICTURE: byte array, the image

The values that will be inserted into the AUTHOR, TITLE and LINK column are parsed from the correctly formatted metadata file. Images in the form of a .jpeg, .jpg and .png file are stored as a corresponding byte array. At the time of retrieval, all byte arrays will be converted to a .png file.

3 Data Understanding

In this section, we will evaluate the initial data we have received to design and test our application with. We will first describe and explore the data and then extend our project plan as a result of this exploration. The project plan will be extended with a concept on which metadata needs to be passed to the database as additional information.

3.1 Data exploration

We expect our users to want to be able to work with different file formats and upload different types of images. Our application should deal with *.jpg*, *.png* and *.jpeg* file extensions. It should also be able to deal with metadata files. We have been provided with a folder of images to use for testing during the development of our application. This folder contains 17 images and 5 metadata files, with the images in formats *.jpg*, *.png* and *.jpeg*, and the metadata files having the extension *.meta*. The metadata files each correspond to one of the images and have the exact same filename as that image. These metadata files contain only the link to the source of the image.

3.2 Project Plan Extension

3.2.1 Mapping Image Metadata to Files

The metadata that we ask our users to add should:

1. allow for easy recovery when querying the database
2. be easy for the user to add
3. be non-mandatory for each field, instead a placeholder should be given

Because of these four requirements, we have chosen to record the following metadata for our images:

1. Author - the author according to the publication. Should be of type String.
2. Title - the title according to the publication. Should be of type String.
3. Link - from the original metadata file (if it exists), type String.
4. Database - the database or website from which the image has been taken, type String.
5. Infographic - what is the image about? Input shall be an integer in range[1,4].
 - (a) a cell/tissue (1)
 - (b) biological cartoon (2)
 - (c) a graph (3)
 - (d) an unknown description (4)

For further considerations on this, please see section 6.3.1.

Originally, we were given a directory in which some images had corresponding metadata files and the others did not. The metadata files that existed contained links related to the image source. We included the links in the modified/new metadata files generated with the help of our command line interface only if the original files existed and contained a link. The link is not taken as

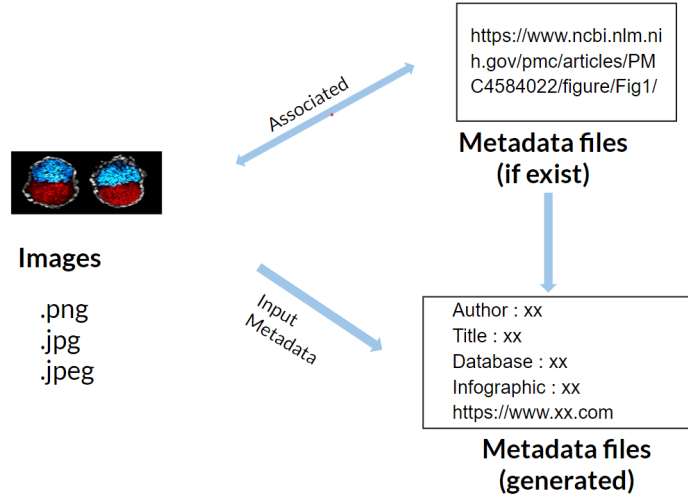


Figure 2: Overview of data understanding process. If the images have associated metadata files then the new input metadata will be appended to the original file. Otherwise, the programme will create a new metadata file itself containing all the input metadata values without the link. In this particular example (left) a picture has an associated metadata file (top right). The original file gets appended with the new input from the user.

an input parameter from the user because entering a link from the command line could cause a lot of errors from the user's side. Other metadata parameters were accepted from the user through the command line interface and written to the (.meta) files, the names of the metadata files were same as the name of the images but with the .meta extension instead of the image extension (see Figure 2).

3.2.2 Creating a Database from Metadata and Image Files

After the mapping of image files to correctly structured and complete metadata files, the process of creating a SQLite database could be started. Each image was treated as a sample and had a unique numerical ID. The information contained in the .txt files was parsed and added to the according columns of AUTHOR, TITLE or LINK (see Figure 3). The superuser can input information into the SQLite Database and also perform queries to extract metadata information or images.

3.2.3 Collaboration

This superuser then might need others to contribute to the database, like colleagues or collaborators on the project the database is needed for. Access to the

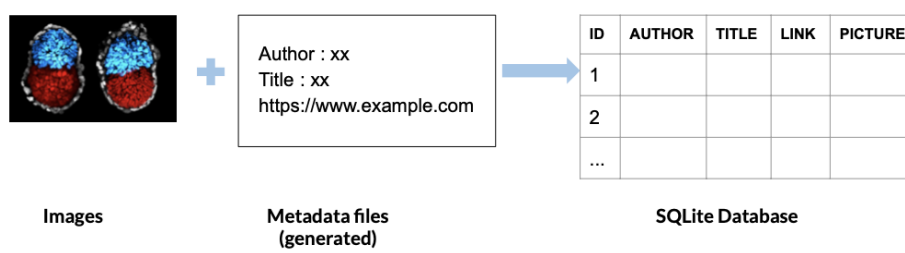


Figure 3: Creating a database and storing images and information from meta-data files.

database should be easy for these users. This is why a created database can be accessed via a RESTful web service, which can easily be accessed via an url that the owner provides to the collaborators. The functionalities on this web service should allow collaborators to contribute effectively, but it should not give them all the options, so that they cannot, for example, accidentally generate empty databases.

The service we will implement will be a RESTful API built using the Spring framework ([Johnson, 2004]). We expect users to interact with the API using the URL provided to them, and to add metadata and file parameters as strings in the command line or into the url itself. The API prototype should work according to the design considerations we set. These are:

1. Superuser
 - (a) flexible set-up
 - (b) limiting options for collaborators
 - (c) privacy
2. Collaborators
 - (a) ease of access
 - (b) ease of use
 - (c) privacy

The Superuser is able to flexibly set up the database where he or she wants and is able to restrict the size of images that collaborators are allowed to upload. He or she is also able to limit the options for collaborators using the `config.json` file, so that they can only access the database they have been given access to. The superusers' privacy is guaranteed by not displaying the full path that the database is saved on, which could contain sensitive information, to the collaborators.

For the collaborators, ease of access is accomplished by providing them with a simple URL to access the database and store and query it. Ease of use is accomplished by only having two simple methods for them to use.

4 Tool building

4.1 Setup of SQLite and Working Environment on Linux

Prior to running our application, you will need to install SQLite on your system. To do this, please follow these steps. 1) Before trying to install, please check whether the installation has already been made by typing into the command line

```
$ sqlite3
```

If SQLite is already installed, you should get the following message:

```
SQLite version 3.29.0 2019-07-10 17:32:03
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

If not, please follow these instructions to install SQLite.

Go to the SQLite webpage (<https://www.sqlite.org/download.html>) and download the most recent version of SQLite-autoconf-*.tar.gz. Type the following commands into the command line to unzip and install the package.

```
$ tar xvfz SQLite-autoconf-*.tar.gz
```

```
$ cd SQLite-autoconf-*
```

```
$ ./configure --prefix = /usr/local
```

```
$ make
```

```
$ make install
```

Confirm successful installation by typing again

```
$ sqlite3
```

In order to set up a working environment several prerequisites have to be fulfilled. For this application the Java Version 1.8.0.231 was used. Apache Maven Version 3.6.3 was installed from <https://maven.apache.org/download.cgi>. Therefore the binaries apache-maven-3.6.3-bin.zip were downloaded.

The following dependencies were added to Maven:

```
SQLite-JDBC Version 3.18.0
commons-cli Version 1.4
commons-io Version 2.6
commons-lang3 Version 3.4
```

by adding the according dependencies to the pom.xml file:

```

<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.18.0</version>
</dependency>

<dependency>
    <groupId>commons-cli</groupId>
    <artifactId>commons-cli</artifactId>
    <version>1.4</version>
</dependency>

<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.6</version>
</dependency>

<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.4</version>
</dependency>

```

The plug-ins of Maven jar plugin Version 1.4 and Maven shade plugin Version 3.2.0 were used.

In order to get the application running you will need to download the repository from gitlab from the following link.

<https://gitlab-sysprog.informatik.uni-bonn.de/ProgrammingLab2/winterterm-2019-20/group-03-descartes/tree/master/ProgrammingProject05>

Please follow these steps:

1. Git clone the repository
2. Open Eclipse
 - (a) Direct the workspace to /group-03-descartes/ProgrammingProject05
 - (b) Import existing maven project
 - (c) Select the de.bit.pl02.pp5.task02 pom.xml for importing the existing maven project
 - (d) Do a maven install
3. After the application is installed you will see a task02-0.0.1-SNAPSHOT.jar in the /target folder
4. To execute the jar use the following command (example in Commands file)

```
java -cp <path-to-jar>/task02-0.0.1-SNAPSHOT.jar
de.bit.pl02.pp5.task02.CommandLineInterface
```

or use the Commands file to run from the base folder from your computer

```
bash /group-03-descartes/ProgrammingProject05/
Task02/task02 Commands.txt
```

5 Implementation

5.1 Task01: Mapping Image Metadata to Files

In this first part of the application a mapping between the image metadata and corresponding files is generated. The user gives the name of the directory from which the files are taken by **-d** or **--directory**. The image which is handled by this command is specified by **-ip** or **--inputfile**. The name of this image file (.jpg) will be used for the corresponding metadata file (.txt) to ensure a correct and unique mapping.

If the user wants to add metadata information to this file, the **-m** or **--meta** flag is used. The exact values to be inserted in the metadata file can be specified by the **-im** or **--inputmeta** option. The input values have to be separated by a comma and follow the order of

author_name , title_name , database_name , infographic_integer

An example input for the terminal can be seen in Figure 4.

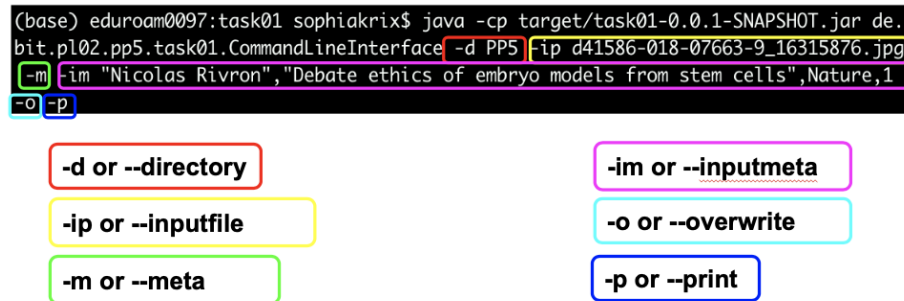


Figure 4: An example input for the terminal to store a file. The directory from which the image and metadata files are taken is specified. Metadata is added to the input file containing information about author, title and infographic type. Below are listed the commands that the user can give in short and long form.

After running this command in the terminal, the user will receive a terminal output as seen in Figure 5 on the left handside. Here, the user is given information about the file name of the image file that is being handled at the

moment and the path of the metadata file. If the corresponding metadata file already exists, it is reported to the user and that this metadata file will be dealt with. Furthermore, the contents of the metadata file that were added are listed. These should now contain the values that were given by the user via the **-im** or **-inputmeta** option. Next to the output at the terminal, the actual metadata file (.txt) is saved at the same location as the input image file. The example output file is displayed in Figure 5, right.



Figure 5: Example output for the terminal (left) and as a metadata file (right).

5.2 Task02: Interaction with the SQLite Database

In the next step, a database is created with the SQLite database engine in which the images and corresponding metadata information will be stored.

In order to create a SQLite database, the user has to provide a directory that contains images and corresponding complete metadata files. The directory path is to be given with the **-d** or **-directory** flag. The database that the user wants to create and connect to again at a later point in time to query has to be uniquely identifiable. Therefore, the user has to input the path where the database is to be saved on the local file system and the name of the database, separated by a comma with the **-n** or **-name** option.

After the database has been created, it is possible to query the database. As an example the option **-git** or **-getImagebyTitle** is used here to extract the image. This example is given in Figure 6. In the following all possible options and search queries will be explained.

The user can search for additional metadata information about an image. Therefore, the user can specify the author by who he wants to search with the **-gma** or **-getMetabyAuthor** option. This requires two values, namely the name of the author and the output path of the metadata file that is to be generated. These two values have to be separated by a comma with no space inbetween (see Figure 7, top).

```
(base) eduroam0097:~ sophiakrix$ java -cp ~/gitlab/group-03-descartes/ProgrammingProject05/Task02/task02/target/task02-0.0.1-SNAPSHOT.jar de.bit.pl02.pp5.task02.CommandLineInterface -d ~/gitlab/group-03-descartes/ProgrammingProject05/Task01/task01/PP5 -n Desktop,trialvy -git "Debate ethics of embryo models from stem cells",Desktop
```

-d or --directory

-git or --getImagebyTitle

-n or --name

Figure 6: Example input to store a directory containing images and corresponding metadata files in a SQLite database. Additional options (blue) can be used to retrieve specified information.

It is also possible to search for metadata by title with the **-gmt** or **getMetabyTitle** option. This option requires similarly two values, the name of the title and the output path of the metadata file, both again separated by a comma (see Figure 7, bottom).

After running this command, the user will get an output at the terminal (see Figure 8, left). This displays information about the name of the database that the user connected to and the elements contained in the database. Below, the contents of the metadata file that is generated here are printed. As there are two images by the title "Debate ethics of embryo models from stem cells" in the database, the two hits are displayed.

Accordingly, the output files are generated containing the metadata information on the images (see Figure 8, right). The name of the output files are generated by combining the name of the author and the id that the image has in the database to create a unique file name.

Instead of retrieving metadata, it is also possible to retrieve the image itself and save it on the local file system. Therefore, option **-gia** or **getImagebyAuthor** can be used. This option requires the arguments of the name of the author and the output path of the image file, separated by a comma (see Figure 9, top). By using the same syntax, the user can also search by title if the **-git** or **getImagebyTitle** flag is used (see Figure 9, bottom).

After running the example command **-git** with the specified title, the terminal output is displayed to the user. It gives information about the database that the user is connecting to and about the exact SQL search query. If the search was successful, the user gets a message about the location where the retrieved images are saved at (see Figure 10, left).

The example images are saved at the specified output path that was given by the user (see Figure 10, right). The names of those images are generated by combining the author name and the id that the image has in the database.

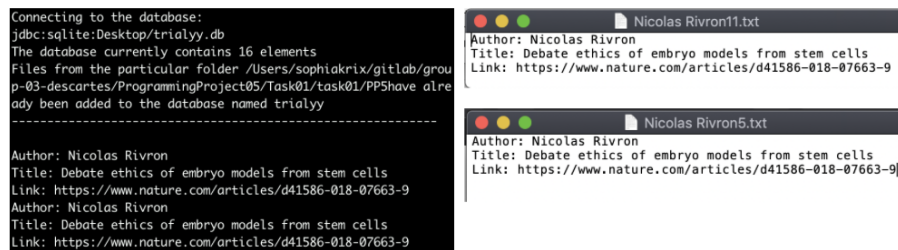
-gma or --getMetabyAuthor: Enter the name of the author of which you want to retrieve the metadata and the outputpath where to save it at

```
-gma author_name,output_path
```

-gmt or --getMetabyTitle: Enter the name of the title of which you want to retrieve the metadata and the outputpath where to save it at

```
-gmt title_name,output_path
```

Figure 7: Options to retrieve metadata information. The user can search by author or title to get additional metadata information of an image. Additionally, the output path of the metadata file that is to be generated has to be specified.



```
Connecting to the database:
jdbc:sqlite:Desktop/trialyy.db
The database currently contains 16 elements
Files from the particular folder /Users/sophiakrix/gitlab/group-03-descartes/ProgrammingProject05/Task01/task01/PPShave already been added to the database named trialyy
-----
Author: Nicolas Rivron
Title: Debate ethics of embryo models from stem cells
Link: https://www.nature.com/articles/d41586-018-07663-9
Author: Nicolas Rivron
Title: Debate ethics of embryo models from stem cells
Link: https://www.nature.com/articles/d41586-018-07663-9
```

```
Nicolas Rivron11.txt
Author: Nicolas Rivron
Title: Debate ethics of embryo models from stem cells
Link: https://www.nature.com/articles/d41586-018-07663-9

Nicolas Rivron5.txt
Author: Nicolas Rivron
Title: Debate ethics of embryo models from stem cells
Link: https://www.nature.com/articles/d41586-018-07663-9
```

Figure 8: Example output for the retrieval of metadata for searching with the **-gmt** option, specifying the title to be searched by. The output of the terminal (left) and the metadata file (right) are shown.

-gia or --getImagebyAuthor Enter the name of the author from which you want the image and the outputpath where to save it at

```
-gia author_name,output_path
```

-git or --getImagebyTitle: Enter the name of the title from which you want the image and the outputpath where to save it at

```
-git title_name,output_path
```

Figure 9: Options to retrieve image files. The user can search by author or title and has to give an output path for the image.

```
Connecting to the database:
jdbc:sqlite:Desktop/trialyy.db
The database currently contains 0 elements
Executing Store method
executed the query:SELECT * FROM IMAGES WHERE TITLE='Debate ethics of embryo model
s from stem cells';
saved image at: Desktop/Nicolas Rivron5.png
saved image at: Desktop/Nicolas Rivron11.png
```

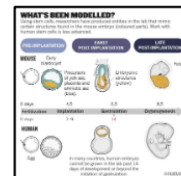
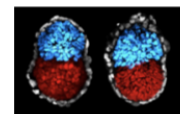


Figure 10: Example output for the retrieval of image files for searching with the **-git** option, specifying the title to be searched by. The output of the terminal (left) and the image file (right) are shown.

5.3 Task03: Web Service

After the user has created the database, he or she might want others to contribute to it. The web service is designed for users that contribute to a database.

5.3.1 Background

For implementing the web service, a RESTful API was chosen. An Application Programming Interface (API) is a communication protocol between parts of a program. It is currently mostly understood and used as a way for clients and servers on the web to communicate ([Braunstein, 2018]). REST is a way to implement an API, and defines a set of constraints that all REST API's should follow to ensure interoperability of these kinds of interfaces on the web. If an interface satisfies these conditions, it can be called a RESTful interface ([Thomas, 2000]).

Spring is a framework to build Java applications. Spring Boot is a part of Spring, and allows users to easily build Java applications. It follows the convention-over-configuration paradigm, which aims to reduce the number of design choices developers have to make without losing flexibility ([Clozel, 2019]).

Using Spring Boot, it is easy to build a RESTful API. Building an application with the Spring Boot framework takes care of most configurations and dependencies, and can be easily built using the Spring Initializr (figure 11).

The RESTful API built using Spring Boot provides an interface between the SQLite database, created by the superuser, and the collaborator at their computer at home or at work (see figure 1). The collaborator can interact with the database via two modalities:

1. command line interface
2. web browser

And can access the following functions:

1. `/store`
2. `/get`

Figures 12 to 16 show how the user can interact with the database using these different modalities and functions. Figure 12 shows how the user can interact with the `/store` method via the command line interface. This functionality allows the user to store an image and associated metadata into the database. It is accessible via the command line tool `curl`, installed by default on most Linux systems. Information to store can be input using the `curl -F` command, followed by the parameter name (e.g. `title`) and the parameter value. There are four parameters that can be added:

1. `title`
2. `author`

3. link

4. file

The first three are metadata parameters and the last one is the filename. Since this last parameter is a file, it needs the decorator @ . In case of parameter values that contain spaces or special characters, the -F statement needs to be within single quotation marks. The parameters are followed by a URL. This URL gives the location of the database, and will be provided by the superuser to the collaborators. For curl to be able to access the file, the user has to navigate to the correct folder in the command line before querying. After sending the request, a response will show, indicating if the file upload has succeeded.

Querying the database is done using the /get method. This method can be used in two ways. Figure 13 shows how to search the database via the command line on author or title. This method returns a list of entries which contain an author and/or title that matches the query. The actual image can be recovered using the /get method with the id keyword, as shown in figure 14. Both these /get methods can also be accessed via the web browser, as shown by figures 15 and 16.

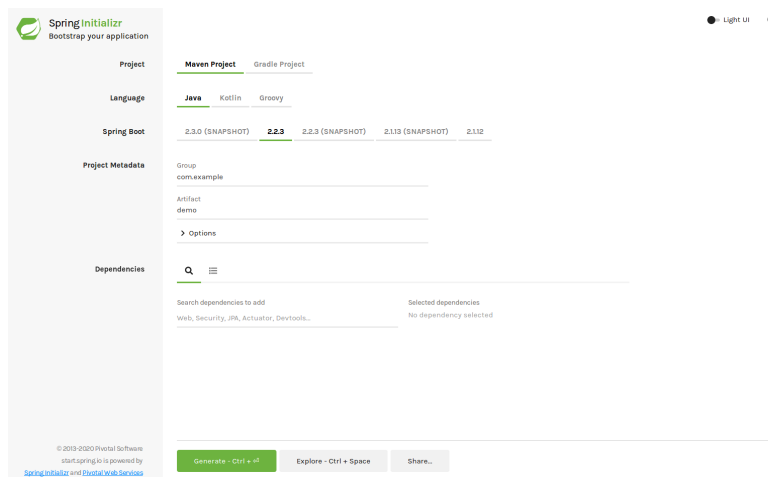


Figure 11: Spring Initializr web interface. This interface allows the user to quickly assemble the needed dependencies to build a Spring Boot application, such as a RESTful API. From [Software, 2013].

```

gemma@maanvis:~/Downloads$ curl -F 'author=Nicolas Rivron' -F 'title=Debate ethics of embryo models from stem cells' -F link=https://www.nature.com/articles/d41586-018-07663-9 -F file=@embryomodelsdebate.png http://localhost:8080/trialvy/store
{"fileName":"You have succesfully uploaded file: embryomodelsdebate.png"}

```

curl command line tool

query

response

Figure 12: Accessing the /store method of the RESTful API using command line tool curl. Curl is called first (in red), then the curl query is given. This query consists of the input parameters author, tile, link and file, followed by the url to access the method. curls response is given in blue.

```

gemma@maanvis:~$ curl http://localhost:8080/trialyy/get?author=Nicolas%20Rivron
{"id":5,"author":"Nicolas Rivron","title":"Debate ethics of embryo models from stem cells","link":"https://www.nature.com/articles/d41586-018-07663-9"}
{"id":11,"author":"Nicolas Rivron","title":"Debate ethics of embryo models from stem cells","link":"https://www.nature.com/articles/d41586-018-07663-9"}
}
gemma@maanvis:~$

```

curl command line tool

query

response

Figure 13: Accessing the /get method for author or title of the RESTful API using command line tool curl. Curl is called first (in red), then the curl query is given. This query consists of the url to access the method. Curls response is given in blue.

```
gemma@maanvis: ~/Downloads$ curl http://localhost:8080/trialyy/get?id=11 -o e
nrvomodelsdebate.png
```

	% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Curr	
ent				Dload	Upload	Total	Spent	Left	Spee
d	0	0	0	0	0	0	--:--:--	--:--:--	
100	461k	100	461k	0	0	1002k	--:--:--	--:--:--	100
2k									

Figure 14: Accessing the /get method for id of the RESTful API using command line tool curl. Curl is called first (in red), then the curl query is given. This query consists of the url to access the method and a filename to save the queried file to. Curls response is given in blue.



Figure 15: Accessing the /get method for author or title of the RESTful API using the web browser.

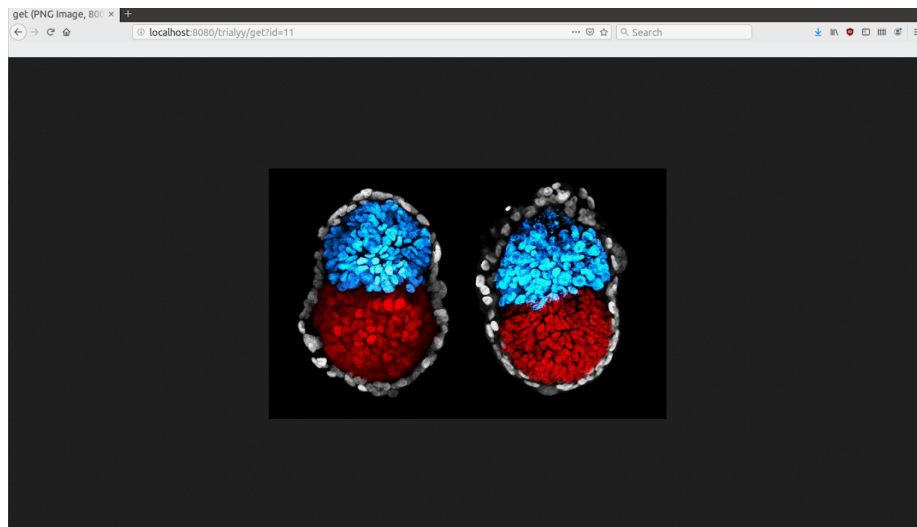


Figure 16: Accessing the /get method for id of the RESTful API using the web browser.

6 Evaluation

In this section we will evaluate the application, as well as the application building process. We will conclude this report with a short outlook, determining the future steps needed to bring this application from prototype to product.

6.1 Result Evaluation

The goal that this application, as defined in section 2.3, was:

The goal of this project is to develop an application, written in the Java programming language by the authors of this thesis in the given time frame, that can create databases, store (folders of) images and metadata files, and allows for collaboration with other researchers.

The developed application meets all the requirements defined in the goal. It can create databases based on the SQLite relational database management system, is able to store images and metadata files, as well as manually add metadata to the database. Storing can be done by the superuser when constructing the database, or by collaborators via a RESTful API. Storing folders of images and metadata files is also possible for the superuser. Collaborators can store single images and metadata information as several string inputs.

6.2 Process Review

The process of building the application was the authors' first real exposure to building a large project in Java. Previously, we were only versed in Python and small Java exercises. During the course of this project, we taught ourselves many skills in both Java and project management.

For project management, gitlab ([GitLab, 2019]) proved essential. We learned how to use this tool to share and collaborate on our code.

We used the integrated development environment (IDE) Eclipse ([Eclipse, 2001]) to write our code in, and Maven ([Maven, 2013]) as our build automation tool.

Through working on this project, we developed an understanding of the suitability of Java when building a readily deployable application. In addition, we learned that the object oriented paradigm it enforces can be hard at first, but finally leads to clearly organised, elegant software solutions.

6.3 Future Steps

During the course of this project, we have successfully developed a prototype application that meets the goal set in section 2.3, and it thus able to be used according to our design considerations. It is, however, a prototype and not yet ready for large scale deployment. This section describes future steps that can or should be taken to improve the application further and allow for its final deployment.

6.3.1 Image and Metadata processing

A first important improvement deals with how the application handles two different images from the same article. Our current application retrieves the author, title and link as additional metadata information. The user is not constrained in what can be filled in at these fields. This could mean that the user chooses as 'title' parameter the title of the whole publication, and as 'link' parameter the link to the whole publication. This would result in entries that are different, but cannot be separated by the metadata. A possible improvement could thus be to restrict the user to not use the exact same metadata as has been added to the database before. Alternatively and less strict, we could publish submission guidelines that warn the user not to use the 'title' and 'link' string that belongs to the whole publication, but the image specifically. A last alternative could be to add the original file name of the image to the metadata we process, since this is likely to be unique.

A second improvement would be to prevent accidental double submission of identical images, since these clutter that database needlessly. A strategy could be to display error messages when the user uploads a file with the same filename as a file already in the database, or a file with the exact same metadata information. The image and metadata information of the image already in the database should then be displayed on request, for example by the user clicking on a string "display duplicate image" in the error message. This way, the user can easily verify if the image she/he wanted to upload was actually a duplicate, or if the filename and/or metadata simply happened to coincide.

A third improvement would be to extend the metadata we store with the images to include an automatically assigned class. This would include a image classification step, as implemented by our colleagues of group 4, Vinay Bharadhwaj, Yojana Gadiya and Madhusanka Tharindu who implemented project 06. Enhancing our application with this functionality would allow users to search using one of the categories as an additional search parameter, which could significantly enhance their search experience.

A fourth improvement could be to revert the image re-formatting currently done by the application. For simplicity reasons when downloading, our current prototype converts all input image format (.png, .jpeg and .jpg) into .png when the image is queried and downloaded. Since different formats are optimal for different types of images ([Team, 2016]), the application could be improved by retaining the original image formats.

Possible other improvements could be to extend the metadata with ontology-controlled keywords describing the image, or to order the search results by relevance, instead of input order as it is now. Searching could also be made more flexible by allowing partial matches or making the search case-insensitive. To address pitfalls of storing whole images as byte arrays in the database one possible way could be connecting web servers to the database, this would allow the database to be linked to images stored elsewhere and eliminate the need to store the pictures in the database.

6.3.2 Application Access

The current prototype is accessible to the superuser via command line options and the collaborator via command line or a web browser. A first improvement could be to extend the functionalities that the API has to allow collaborators to also upload images and metadata files in batch mode, instead of one by one.

A second improvement would be to build a front-end for our application. Currently, users can interact with the database using the url they were provided via the web browser, but only for the `/get` functionality. They are able to query the database via the browser, but can only store images using the command line. A simple front-end that allows form input and file upload, connected to the API controller class, would allow users to also use the `/store` functionality via the web browser, which would greatly improve the ease of use for collaborators. This front-end could then also be extended to make `/get` requests more user-friendly, by adding a form input to the page. This way, queries would not have to be made by changing the url directly, but could be made by for example clicking on the metadata parameter you would like to search by and adding the query in the text box next to it on the page.

An additional benefit of adding a front-end would be in displaying the results of your query in a nicer format. Currently, `get` requests that query by author and title display their result in a simple JSON format on the page. This could be improved using a front-end that displays these results in a more pleasing way. A valuable addition would then be to make every single result clickable. Currently, only the `'link'` parameter is clickable, and will take the user to the original source of the image. If the `'id'` parameter were clickable, it could take the user directly to that image, without having to do a second `/get` request. If the `'author'` or `'title'` parameters were clickable, the user could be taken to another `/get` request that searched for all information pertaining to the clicked author or title.

If a front-end were implemented, we could then further experiment with how users experience certain other configurations. For example, a search by author or title could give a small thumbnail of the image, next to the id and metadata information. This would possibly allow the user find the requested picture fast, since they have a visual preview. It could however lead to cluttering of the search results window, causing the user to need to scroll down much more. Another possible feature would be to add metadata information to the `get` by id functionality. This method would then not only display the image but also its associated metadata in the same screen. For this display to look attractive, all images should be displayed at the same size.

A third improvement could be to display more and more informative error messages to the collaborators. Currently, the application throws an error when the collaborator tries to access a database that has not been configured. When using the application via the command line, `curl` also throws errors when syntax mistakes are made or when the user is about to display an image directly in the command line. Error messages could be added that warn the user when they are searching for an id, author or title that does not yet exist. Currently, the

application then simply returns no search results.

An additional warning could also be displayed to the superuser when they are adding a non-unique database name to the config.json file. Since this name is used as a key in a HashMap object in Java, the first database name would be overwritten by the second. This could make a database inaccessible, even if it is on a unique path on the server. This is currently a silent fail, since images can simply be added to later database. This silent fail should definitely be fixed before the applications deployment.

A final improvement would be to deploy the API on a proper server for testing and see if any bugs occur there. This server should be accessible to all collaborators, not only the superuser. This is an essential step for effective collaboration.

7 Deployment

Deployment can be possible once all the improvement options as discussed in section 6 have been carefully considered and then either implemented or discarded. Once this is done, the application will first be tested by prospective (beta) users. If these users run into problems, we will improve our application and release a new version, which new beta users will test. This process will continue until the application is at an appropriate level. We will then publish our application for use by all users. We will continue monitoring and reacting to issues our users might have.

A Approach: Adapted CRISP-DM

In order to develop an application that fit the users needs and was designed and implemented in a way corresponding to industry standards, we choose to use the CRISP-DM model to guide our development ([Shearer, 2000]).

We adapted the CRISP-DM process model ([Chapman et al., 1999]) to our purposes, changing the standard slightly to fit the tool building nature of our project, in contrast to the data mining projects it was designed for. This is the original structure of CRISP-DM:

1. Business Understanding
 - (a) Determine Business Objectives
 - (b) Situation Assessment
 - (c) Determine Data Mining Goal
 - (d) Produce Project Plan
2. Data Understanding
 - (a) Collect Initial Data
 - (b) Describe Data
 - (c) Explore Data
 - (d) Verify Data Quality
3. Data Preparation
 - (a) Select Data
 - (b) Clean Data
 - (c) Construct Data
 - (d) Format Data
 - (e) Integrate Data
4. Modeling
 - (a) Select Modelling Technique
 - (b) Generate Test Design
 - (c) Build model
 - (d) Assess modelling
5. Evaluation
 - (a) Evaluate Results
 - (b) Review Process
 - (c) Determine Next steps

6. Deployment
 - (a) Plan Deployment
 - (b) Plan monitoring and Maintenance
 - (c) Produce final report
 - (d) Review Project

This is the structure we adapted it to:

1. User Understanding (was Business Understanding)
 - (a) Determine User Objectives
 - (b) Situation Assessment
 - (c) Determine Application Goal
 - (d) Produce Project Plan
2. Data Understanding
 - (a) Describe and Explore Data
 - (b) Extend project plan for data
3. Tool Building (was Modeling)
 - (a) Select Tools
 - (b) Build Application
 - (c) Test Application
4. Implementation
5. Evaluation
 - (a) Evaluate Results
 - (b) Review Process
 - (c) Determine Next steps
6. Deployment
 - (a) Plan Deployment
 - (b) Plan monitoring and Maintenance
 - (c) Produce final report
 - (d) Review Project

The data Preparation stage is omitted entirely because our application should not work for a specific set of data.

This thesis will follow the steps described above to guide the reader through our design process.

References

- [Braunstein, 2018] Braunstein, M. L. (2018). *Health Informatics on FHIR: How HL7's New API is Transforming Healthcare*. Springer.
- [Chapman et al., 1999] Chapman, P., Clinton, J., Kerber, R., Khabaza, T., Reinartz, T., Shearer, C., and Wirth, R. (1999). The crisp-dm user guide. In *4th CRISP-DM SIG Workshop in Brussels in March*, volume 1999.
- [Clozel, 2019] Clozel, B. (2019). Spring framework 5.2.2 and 5.1.12 available now. <https://spring.io/blog/2019/12/03/spring-framework-5-2-2-and-5-1-12-available-now>. Accessed: 2020-01-21.
- [Eclipse, 2001] Eclipse (2001). Eclipse tools project. <https://archive.eclipse.org/eclipse/downloads/drops/R-1.0-200111070001/>. Accessed: 2020-01-21.
- [GitLab, 2019] GitLab (2019). History of gitlab. <https://about.gitlab.com/company/history/>. Accessed: 2020-01-21.
- [Hipp,] Hipp, R. Sqlite.
- [Johnson, 2004] Johnson, R. (2004). *Expert one-on-one J2EE design and development*. John Wiley & Sons.
- [Maven, 2013] Maven, A. (2013). Apache projects release listing. <https://projects.apache.org/releases.html/>. Accessed: 2020-01-21.
- [Shearer, 2000] Shearer, C. (2000). The crisp-dm model: the new blueprint for data mining. *Journal of data warehousing*, 5(4):13–22.
- [Software, 2013] Software, P. (2013). Spring initialzr. <https://start.spring.io/>. Accessed: 2020-02-05.
- [Team, 2016] Team, E. (2016). 1st web designer, understanding the most popular image file types and formats. <https://1stwebdesigner.com/image-file-types/>. Accessed: 2020-01-21.
- [Thomas, 2000] Thomas, R. (2000). Fielding. chapter 5: Representational state transfer (rest). *Architectural Styles and the Design of Network-based Software Architectures (Ph. D.)*.

B Statement of Authorship

We hereby declare that we are the sole authors of this thesis and that we have not used any sources other than those listed in the bibliography and identified as references.

Bonn, February 7, 2020

Shreya Kapoor
Sophia Krix
Gemma van der Voort