xrf-explorerV2

FILE: XRF_ExplorerV2_softwaredesigndocument.pdf

General information on the software documentation
version: 2024 October 02

---

xrf-explorerV2 is a research software toolkit developed by students at Eindhoven University of Technology (TU/e), for the Van Gogh Museum Amsterdam in a partnership with ASML. The toolkit is intended to be used for the integrated exploration of multimodal images, spectral data, and chemical mappings of a painting. The toolkit is developed to assist in the conservation science practice.

Project

The original prototype (V1) was created by a TU/e master's student from January–September 2023. The current version (V2) was developed by a team of TU/e bachelor students from April–July 2024.

Documentation

Five documentation files were delivered by the development team.
Together they form a comprehensive documentation package:

- XRF_ExplorerV2_userrequirementsdocument.pdf
- XRF_ExplorerV2_softwaredesigndocument.pdf
- XRF_ExplorerV2_acceptancetestplan.pdf
- XRF_ExplorerV2_softwaretransferdocument.pdf
- XRF_ExplorerV2_softwareusermanual.pdf

Acknowledgements

For the development, testing, and documenting of xrf-explorer, test data was provided by the Museum of Modern Art (New York) to the development team. It consisted of images and xrf scanning data of the "Portrait of Joseph Roulin" painting by Vincent van Gogh. Images of this painting are included in some of the documentation files for the purpose of explaining the software usage.

For more detailed information on this painting, including image licenses, we refer to: https://www.moma.org/collection/works/79105

License

Software

The xrfexplorerV2 software itself (codebase and code documentation) can be found at the github repository "Olive-Groves/xrf-explorer" (see: https://github.com/olive-groves).

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Software Design Document

2IPE0 SOFTWARE ENGINEERING PROJECT

July, 2024

GROUP 6

PROJECT TEAM

Adrien Verriele	1710303
Diego Rivera Garrido	1674196
Dirk Burgers	1653873
Iliyan Teofilov	1671952
Ivan Ivanov	1661469
Jan Bulthuis	1696866
Lotte Lakeman	1668137
Massimo Leal Martel	1662147
Pablo Benayas Penas	1667939
Ruben Savelkouls	1695347
Sonia Maxim	1675656

PROJECT MANAGERS

Antreas Efstathiou

Konstantinos Chanioglou

SUPERVISOR

Prof dr. R. Bloo

CLIENTS

Ana Martins

Lars Maxfield

Marco Roling

ABSTRACT

This document is the Software Design Document (SDD) for XRF Explorer 2.0, a web application, which facilitates the analysis of paintings' chemical composition for conservation scientists. This product has been developed by a group of students at the Eindhoven University of Technology as part of their Software Engineering Project with the guidance of the Van Gogh Museum and ASML, acting as the client. The goal is to deliver an application allowing conservation scientists to perform in-depth analyses of several types of scans of a painting, most notably X-Ray Fluorescence mappings. The design decisions presented in this document comply with the requirements that have been stated in the User Requirements Document (URD).

Contents

Document Status Sheet 6

Document Change Records 8

1 Introduction 9

1.1 Purpose 9

1.2 Scope 9

1.3 List of definitions 9

1.3.1 Terms 9

1.3.2 Acronyms and abbreviations 10

1.4 List of references 10

1.5 Overview 11

2 System Overview 12

2.1 Relation to current projects 12

2.2 Relation to predecessor and future projects 12

2.3 Function and purpose 12

2.4 Environment 14

2.5 Relation to other systems 15

3 System Architecture 16

3.1 Architectural design 16

3.1.1 Global architecture design 16

3.1.2 Front-end 16

3.1.3 Back-end 19

3.2 Logical model description 19

3.2.1 Front-end 20

3.2.2 Back-end 33

3.3 System dynamics 44

3.3.1 Upload a data source 44

3.3.2 Delete a data source 48

3.3.3 Add data to existing data source 49

3.3.4 Delete data from existing data source 50

3.3.5 Edit data source elements 51

3.3.6 Edit the elemental channels of existing data source 52

3.3.7 Load a data source 53

3.3.8 Enable the second main viewer 54

3.3.9 Disable the second main viewer 55

3.3.10 Panning in the main viewer 56

3.3.11 Zooming in the main viewer . . . 57
3.3.12 Export main viewer . . . 58
3.3.13 Export a visualization . . . 59
3.3.14 Make lasso selection . . . 60
3.3.15 Make rectangle selection . . . 61
3.3.16 Enabling a layer . . . 62
3.3.17 Changing the layer hierarchy . . . 63
3.3.18 Change the opacity of a layer . . . 64
3.3.19 Change the filter value of a layer . . . 65
3.3.20 Resetting the filter value of a layer . . . 66
3.3.21 Enabling the lens and choosing a layer . . . 67
3.3.22 Locking the lens . . . 68
3.3.23 Generate spectra chart of the global average . . . 69
3.3.24 Generate theoretical spectra chart of a selected element . . . 70
3.3.25 Generate spectra within selection . . . 71
3.3.26 Generate bar chart with the average abundance of every element . . . 72
3.3.27 Generate line chart with the area selected by the user . . . 73
3.3.28 Generate embedding using dimensionality reduction . . . 74
3.3.29 Showing dimensionality reduction embedding with overlay . . . 75
3.3.30 Highlighting the selection in the dimensionality reduction window . . . 76
3.3.31 Generating color segmentation with user-given parameters . . . 77
3.3.32 Enabling an elemental channel . . . 78
3.3.33 Changing elemental channel highlight color . . . 79
3.3.34 Changing elemental channel highlight intensity . . . 80
3.4 Data model . . . 81
3.5 External interface definitions . . . 81
3.5.1 Interaction with the user’s file system: . . . 81
3.5.2 Interaction with the user’s browser: . . . 81
3.5.3 Interaction with the server’s file system . . . 82
3.6 Design rationale . . . 82
3.6.1 Vue.js . . . 82
3.6.2 Image Viewer . . . 83
3.6.3 D3.js . . . 83
3.6.4 Python . . . 83
3.6.5 Flask . . . 83

4 Feasibility and Resource Estimates **84**
4.1 Hardware requirements . . . 84
4.1.1 Hardware testing . . . 84
4.1.2 Server requirements . . . 85
4.1.3 Client machine requirements . . . 85
4.1.4 Software requirements . . . 86
4.2 Performance metrics . . . 86
4.2.1 Project creation . . . 86
4.2.2 Project startup . . . 86
4.2.3 Spectral charts global average . . . 86
4.2.4 Spectral charts selection average . . . 87
4.2.5 Elemental charts global average . . . 87
4.2.6 Elemental charts selection average . . . 87
4.2.7 Dimensionality reduction . . . 87
4.2.8 Color segmentation . . . 88

A Front-end Class Diagram **89**

DOCUMENT STATUS SHEET

General

Document title	Software Design Document
Document identifier	SDD/1.0
Document authors	Dirk Burgers Ivan Ivanov Jan Bulthuis Lotte Lakeman Massimo Leal Martel Pablo Benayas Penas Sonia Maxim
Document status	Final

Document history

Version	Date	Authors	Reason
0.01	02-06-2024	Pablo Benayas Penas	Writing the abstract and first version of the introduction.
0.02	03-06-2024	Pablo Benayas Penas	Finalizing first version of the introduction.
0.03	04-06-2024	Pablo Benayas Penas	First version of section 3.
0.04	05-06-2024	Pablo Benayas Penas	Initial version of sequence diagram section.
0.05	16-06-2024	Lotte Lakeman Sonia Maxim	Initial version of the front-end class diagram and explanation. Started on the initial version of the back-end class diagram.
0.06	20-06-2024	Lotte Lakeman	Continued working on front-end class diagram and its explanation.
0.07	22-06-2024	Dirk Burgers Ivan Ivanov Lotte Lakeman	Added sequence diagrams and section 2.
0.08	25-06-2024	Massimo Leal Martel Sonia Maxim	Worked on back-end class diagram and explanation.
0.09	26-06-2024	Dirk Burgers Lotte Lakeman	Worked on sequence diagrams and finalizing section 2.
0.10	26-06-2024	Pablo Benayas Penas	Added class diagram to section 3.
0.11	28-06-2024	Pablo Benayas Penas	Added sections 3.4 and 3.5.
0.12	28-06-2024	Massimo Leal Martel	Finalized back-end class diagram and explanation.
0.13	01-07-2024	Dirk Burgers Lotte Lakeman	Worked on section 3.
0.14	01-07-2024	Massimo Leal Martel	Worked on sections 3.4 and 3.5
0.15	02-07-2024	Dirk Burgers Ivan Ivanov	Added final sequence diagrams to section 3.

Continued on next page

Version	Date	Authors	Reason
0.16	02-07-2024	Diego Rivera Garrido Lotte Lakeman	Added and finalized section 4.
0.17	03-07-2024	Jan Bulthuis	Added section 3.6
1.0	04-07-2024	Jan Bulthuis Iliyan Teofilov Massimo Leal Martel	Final draft.

DOCUMENT CHANGE RECORDS

Version	Date	Section	Reason
0.01	02-06-2024	Entire document	Creation of the document
0.03	04-06-2024	Section 3	Initial version of section 3
0.05	16-06-2024	Section 3	Added class diagrams
0.07	22-06-2024	Section 2	Initial version
0.15	02-07-2024	Section 3	Finalized adding sequence diagrams
0.16	02-07-2024	Section 4	Added and finalized section 4
1.0	04-07-2024	Section 3	Finalized architecture discussion

1 INTRODUCTION

1.1 Purpose

The SDD depicts a high-level overview of the system design and the architecture of the XRF Explorer 2.0. The document explains the rationale behind the design decisions taken during development and the resources needed to run the XRF Explorer 2.0. This document is intended for professionals in the field of software development.

1.2 Scope

The XRF Explorer 2.0 is developed by a group of Bachelor Computer Science and Engineering students for the Software Engineering Project at the Eindhoven University of Technology, in collaboration with the Van Gogh Museum and ASML. This project aims at creating an accessible web application for conservation scientists to explore paintings by comparing high resolution RGB images with a respective X-Ray fluorescence scan, as well as other contextual sources such as UV or X-Ray imagery, all in one place. This project builds upon the 2023 Data Science and Artificial Intelligence Master's project of Dominique van Berkum, also in collaboration with Eindhoven University of Technology, the Van Gogh Museum and ASML [1], who developed the prototype XRF Explorer. The goal is to transform this proof of concept into a robust, documented and maintainable application, capable of running on a live server.

With the XRF Explorer 2.0, conservation scientists can explore several representations of the same painting, such as the original RGB image, the X-Ray fluorescence mapping, UV or the X-Ray scans, and directly compare these against each other to gain additional insight into the making of the painting, allowing for more appropriate preservation techniques. This is achieved by overlaying them in an image viewer, with the option to select only a portion of the painting, for example by manual pixel selection, element presence or color. The application also provides additional methods to visualize the high-dimensional elemental or spectral energy data in the form of interactive charts and dimensionality reduction views.

1.3 List of definitions

1.3.1 Terms

Term	Description
Bitmasks	The bitmasks representing the correspondence of pixels to the different Selection mask. A two-dimensional boolean matrix utilized to specify selected pixels within an image. Each element of the matrix corresponds to a pixel in the associated image and is set to true if the pixel is selected and false otherwise.
Clients	Ana Martins, Lars Maxfield, and Marco Roling.
Color clusters	The colors resulting from applying the color segmentation algorithm to the whole RGB image/per element.
Color segmentation window	Visualization view that displays the distribution of color segments throughout the painting.
Contextual image	Image displaying the painting in a modality, e.g. RGB images or UV images.
Data source	Collection of raw, processed and contextual data that correspond to the same painting and are used to analyze said painting. Note: in the final software these are referred to as **projects**.

Continued on next page

Term	Description
Dialog	A dialog is a small, temporary window that appears on the screen to facilitate user interaction, often requiring user input or action before it can be dismissed.
Dimensionality reduction	Transformation of data from high-dimensional space into low-dimensional space.
Dimensionality reduction window	Window displaying the dimensionality reduction results to the user.
Elemental channel window	Window that displays the elemental channels.
Elemental composition	Abundance and distribution of the elements present in the painting.
Elemental distribution map	Image derived from the processed data, visualizing the distribution of one element across the painting.
Elements	Chemical elements present in the painting.
Embedding	The output resulting from the application of a dimensionality reduction method to a dataset is referred to as an embedding.
Filters	Visual effects applied to a layer visible in the main viewer (includes contrast, saturation, gamma and brightness).
Layer	Discrete compound that contains an individual contextual image or elemental distribution map.
Polygon selection tool	Selection tool which allows for creation of polygonal shapes (both for the dimensionality reduction view and the main viewer).
Elemental/Processed data	Elemental/Processed datacube (3-dimensional) of elemental distribution data obtained from processing the raw data that, for each pixel, gives the abundance of the different elements present.
Raw/Unprocessed data	Raw datacube (3-dimensional) of spectral data obtained from the XRF scanner that, for each pixel, gives the intensity of the X-ray fluorescence emitted at different energies the elements in the painting.
Rectangle selection tool	Selection tool which allows for creation of rectangular shapes (both for the dimensionality reduction view.
Spectra	The range of frequencies of electromagnetic radiation absorbed or emitted by a substance, used to analyze and determine its chemical composition.

1.3.2 Acronyms and abbreviations

Term	Description
SDD	Software Design Document
CS	Color segmentation

1.4 List of references

[1] D. V. B. van Berkum, “Xrf-xplorer: An interactive visual exploration tool for micro-x-ray fluorescence scanning data on paintings,” Master’s thesis, Eindhoven Univerisity of Technology, September 2023.

[2] A. Popa, “Visualayered: Combined visual analysis of ma-xrf and ris data andra popa,” Master’s thesis, Delft University of Technology, December 2022.

[3] L. Maxfield, “Butterfly registrator.” `https://github.com/olive-groves/butterfly_registrator`”.

[4] “OpenSeaDragon.” `https://openseadragon.github.io/`.

[5] “Viewer.js.” `https://github.com/fengyuanchen/viewerjs`.

[6] “IIPMooViewer.” `https://iipimage.sourceforge.io/documentation/iipmooviewer`.

1.5 Overview

In the rest of this SDD, we first describe an overview of the system by comparing it to its predecessor, current and successor projects, to compare it to existing solutions and explain how it can be developed further in the future. We also explain the primary purpose of the system, as well as its usage environment, in terms of hardware, operating system and users.

We then describe the architecture of the system, as well as its logical model description, each supported by class diagrams to explain the components and their relationships. After that, the most important methods of each component are described.

Moreover, we explain the dynamics of the system, including its different transitions, in reaction to events, supported by sequence diagrams. Finally, definitions of interfaces with external systems, and a rationale for the architecture design and implementation choices are given.

2 SYSTEM OVERVIEW

In this chapter, we compare the XRF Explorer 2.0 relative to its predecessor and alternative projects, and discuss its future development, as well as its purpose and its target environment.

2.1 Relation to current projects

The XRF Explorer 2.0 addresses the limitations observed in the existing state-of-the-art tools and applications that target the exploration of XRF data in paintings. Currently, conservation scientists use a variety of software solutions for analysis of paintings that only target specific needs. These tools usually consist of basic representations of individual elemental maps and interactive image viewers. These solutions do not fully exploit the potential of XRF data, such as exploring the relations between the elemental compositions.

VisuaLayered [2], a similar tool by Andra Popa, is able to demonstrate the use and potential of linked views in the analysis. However, this tool has its own limitations, particularly in its use of elemental maps without considering the unprocessed data cube of XRF data. Furthermore, VisuaLayered is not readily available to conservation scientists.

The XRF Explorer 2.0 application builds upon the current efforts and gained insights and allows the analysis of both processed and unprocessed data, in relation to contextual data such as RGB, UV and X-Ray images. It aims to provide an integrated environment that combines the strengths of all individual data sources, with a user-friendly interface that is accessible to conservation scientists.

2.2 Relation to predecessor and future projects

The concept of the XRF Explorer 2.0 application is based on the prototype introduced by Dominique van Berkum in his master's Thesis, which main focus is the exploration phase of the Micro-XRF data [1]. The XRF Explorer 2.0 does not build directly on the codebase of this thesis project. Instead, it uses its fundamental ideas to create a robust and maintainable implementation with enhanced features and an improved user experience. The XRF Explorer 2.0 is thus a project which has been built from the ground up, but with the main research and conceptualization already being done by its predecessor.

The application will be made available under an open-source license. Together with the robust implementation and thorough documentation of the project, this allows for future projects to easily build upon the basis of this application. Eventually, ASML's intention is to take over and expand upon the project.

2.3 Function and purpose

The purpose of the XRF Explorer 2.0 is to allow conservation scientists to view and explore painting data in a standardized environment. The visual data of paintings gets lost over time as pigments fade, however, the elemental composition does not. By analysing the elemental composition of paintings, conservation scientists are able to restore paintings to their original state. The final implementation of the XRF Explorer 2.0 has a large selection of functionalities allowing for conservation scientists to freely explore painting data.

Create and edit data sources

The XRF Explorer 2.0 allows users to create new data sources of paintings they wish to explore. Additionally, if new data is obtained on an already existing data source, the XRF Explorer 2.0 allows users to edit and add additional data to already existing data sources.

Main viewer

Perhaps the most important component is the application's large main viewer. In this main viewer, the user can freely pan around their uploaded painting and zoom in on details. With the usage of WebGL, this is a highly optimized process allowing for large images to be rendered with ease. Additionally, the user can enable a second main viewer to compare different areas of the painting side-by-side.

Selection tool

The XRF Explorer 2.0 has a rectangle and polygon selection tool. Using these selection tools, the user is able to make a selection in the main viewer. This selection is then highlighted in the main viewer and the data within this selection can be visualized through the spectrum and line charts the XRF Explorer 2.0 supports.

Layer system

The XRF Explorer 2.0 has an intricate layer system. Thanks to the layer system, the user can easily reorder the various images they have uploaded to a data source. The layer system allows users to toggle the visibility of layers, change the layer hierarchy, and change the opacity for specific layers allowing for quick comparison between different data types.

Filters

From the layers window, the user is able to change the contrast, saturation, gamma correction, and brightness of the different layers. These additional filters allow for a customizable view. Combined with the layer system and its opacity slider, these filters greatly support the data exploration process.

The lens

For further exploration in the main viewer, the user can choose a specific layer to be displayed in the lens. When hovering over the main viewer with the lens enabled, the selected layer is available through it allowing for direct comparison between layers (displayed in Figure 1). The user can also lock the lens in place and then freely move their cursor around in other windows.

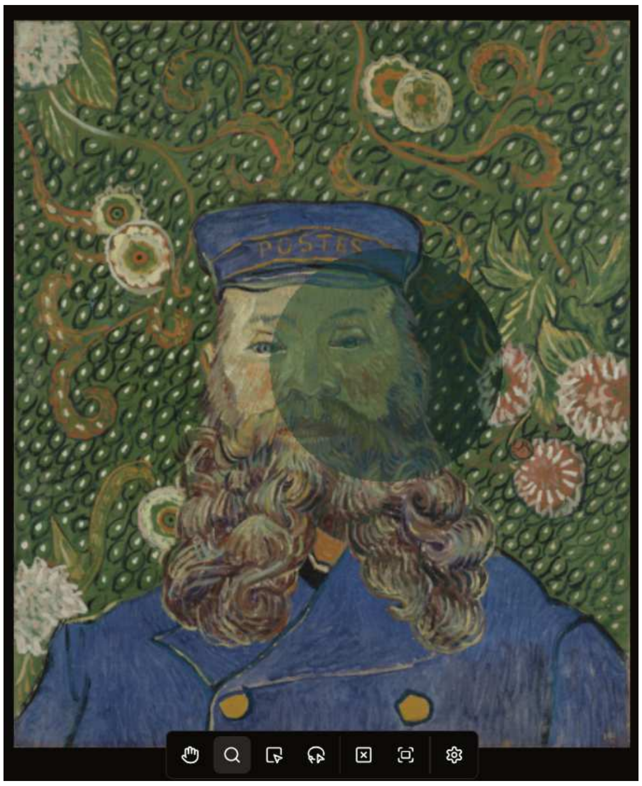

Figure 1: The lens

Elements selection

Within the application, the user is able to select elements they wish to be highlighted in the main viewer. In doing so, the user can better visualize the presence and relation between different elements. To perfect this highlighting functionality, users are able to modify the highlighting threshold and color for each element independently.

Color segmentation

The XRF Explorer 2.0 has a color segmentation window. From this window, users are able to select color clusters (clusters of pixels that are similar in color) to be highlighted in the main viewer. These color clusters can be computed per element or over the painting as a whole.

Dimensionality reduction

Thanks to the dimensionality reduction window, the user has a clear overview of the elemental similarities and differences in composition across specific regions of a painting. From within the generated dimensionality reduction image, the user is able to select specific regions which will then be highlighted in the main viewer. This allows for users to directly see where a part of a mapping can be found within the painting.

Spectrum chart

The XRF Explorer 2.0 has a window for visualization of the spectral energy of the painting. Within this window, the user can view the global spectrum of the painting, as well as the theoretical emission of a chosen element. Additionally, the user is able to visualize the average spectrum over the current selection made in the main viewer.

Bar and line charts

The application has the option to view the elemental abundance as a bar and line chart. The bar chart displays the average elemental composition over the whole painting. The line chart, which is plotted based on the average over the selection, allows the user to directly compare this average to the current selection.

Export images

Plots and images created by using the XRF Explorer 2.0 may be beneficial for academic papers or presentations. To cater to this demand, the application allows the user to export the following images: the main viewer, the dimensionality reduction embedding image, the bar-and-line chart, and the spectrum chart.

2.4 Environment

The XRF Explorer 2.0 was designed as a web application upon request by the client. The client's preference was to avoid executable installations. Additionally, the client specified that the software should be usable regardless of operating system and the application should be usable on the latest version of any browser. Based on these requirements, the following environmental requirements were formulated:

- The user uses one of the following browsers: Chrome (version 124 or higher) or Firefox (version 125 or higher);
- The user uses one of the following operating systems on an x86-64 system: Windows 10, Windows 11 or Ubuntu LTS 24.04.
- The server uses the Ubuntu LTS 24.04 operating system.

Due to the difficult nature of testing on Apple devices, requirements regarding this were omitted. It is, however, believed that the application would correctly run on these devices.

Regarding hardware, due to the nature of the application, RAM would mostly be a concern. Upon discussion with the clients, it was decided that the server would be acceptable to have 16GB RAM as the minimum RAM requirement. Nowadays, this is a reasonable lower bound and testing shows that the application (most importantly the main viewer) runs smoothly with this hardware constraint in place.

Additionally, while discussing the project with the clients, it was decided that user roles are currently not a concern. The final application thus features no user roles and all interactions within the client are available to any user accessing the software.

2.5 Relation to other systems

The XRF Explorer 2.0 is a standalone software and is not part of a larger system. It replaces the proof of concept (version 1) as created by Dominique van Berkum for his master's Thesis [1]. The code of the original software as created by Dominique van Berkum was not used for the final product, the XRF Explorer 2.0.

3 SYSTEM ARCHITECTURE

3.1 Architectural design

In the following section, an abstracted view of the XRF Explorer 2.0 will be covered to provide context on the chosen architecture. The front-end and the back-end will be discussed separately, though the overarching system will also be covered, showing how the two entities communicate.

3.1.1 Global architecture design

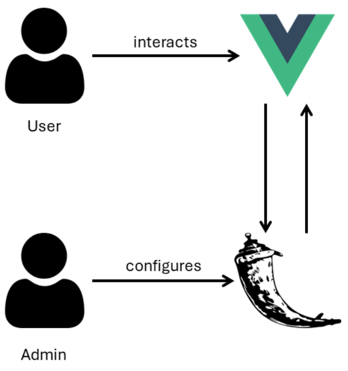

Figure 2: High-level overview of the global architecture

The XRF Explorer 2.0 is a web application developed using Vue.js and Python's Flask framework for the front-end and back-end respectively. The architectural design for each of these entities will be described more thoroughly throughout the rest of Section 3.1. Part of the application's functionality resides in the front-end in order to increase responsiveness for the user. However, due to the large size of the data being handled, large parts of the functionality regarding data processing is delegated to the back-end, where the data is also stored. This is also illustrated in Figure 2.

In essence, the user will interact with the system through the front-end UI, where they will be able to explore the data. The Vue.js framework will handle uploading or querying the necessary data to and from the back-end server, which will respond through the Flask framework. The server administrator can also configure several aspects of the server, such as directory locations, as well as computation parameters.

3.1.2 Front-end

For the sake of simplicity and accessibility to the user, the XRF Explorer 2.0 application contains only one page, which consists of three main components, as seen in Figure 3. The application has a main viewer, a header and multiple windows which can be visualized through the side bars of the application.

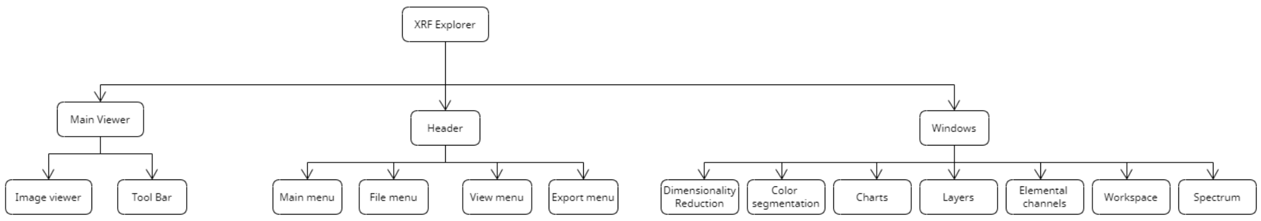

Figure 3: The front-end architecture

The specifics of the various windows and menus are stated in sections 2.3 and 3.1.

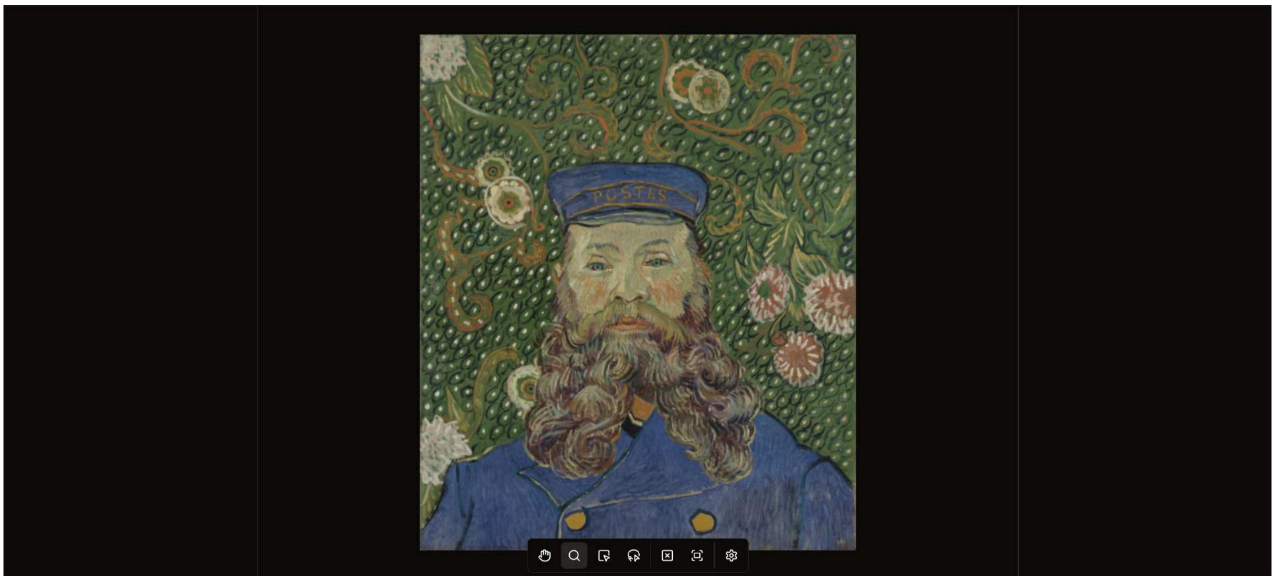

Figure 4: The sidebars of the application on the left and the right

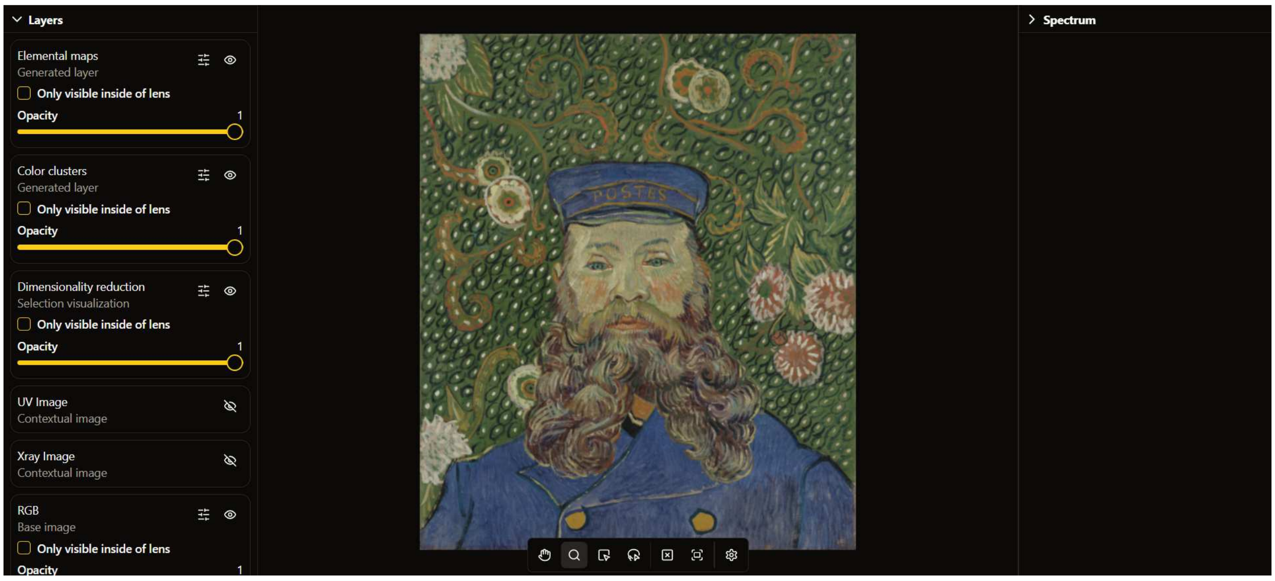

Figure 5: The sidebars with an enabled and a collapsed window

Main viewer

Figure 6 displays the main viewer, which contains the painting image. The user is able to zoom, pan, and make selections within this main viewer. This can be done via the toolbar at the bottom. The toolbar has the following options (from left to right in Figure 6): pan tool (default), lens tool, rectangle selection tool, polygon selection tool, clearing the current selection, resetting the painting to the default position and zoom level, and finally the settings for changing the pan and zoom speed as well as the lens size.
For a painting to be displayed, it needs to be loaded in from the back-end to the front-end. The image viewer makes use of WebGL. Regardless of whether a data source is loaded in, the main viewer is always displayed.

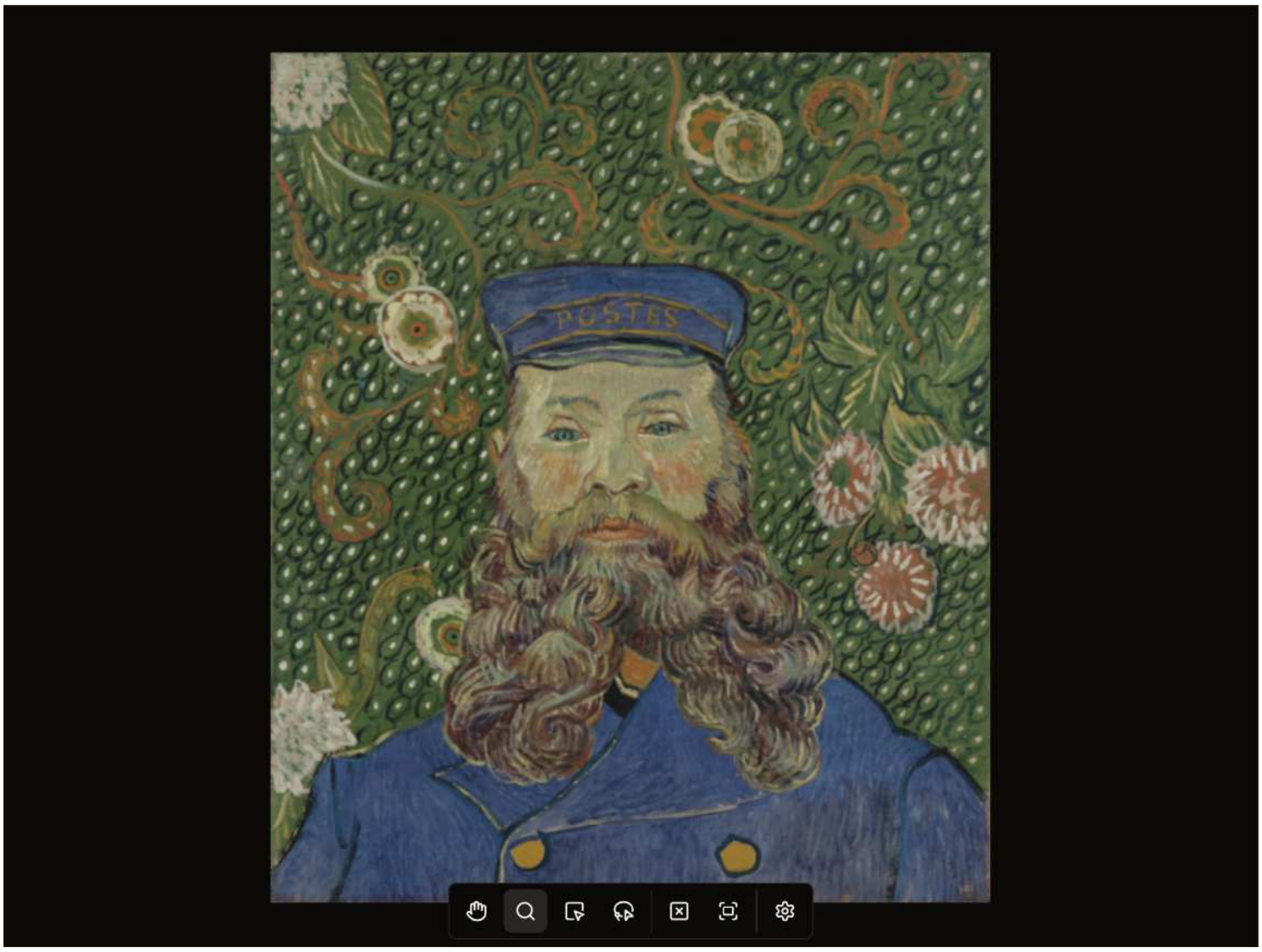

Figure 6: The main viewer of the application and the toolbar

Figure 7: The header of the application

Header

In Figure 7 the header is displayed. From the header, the user can choose the application's theme, access the documentation or the GitHub repository, or reset the client through the main menu (the "XRF-Explorer" button). Furthermore, windows can be enabled/disabled through the "View" button and exported through the "Export" button. From the "File" button menu, new data sources can be uploaded and existing ones can be accessed.

Windows

In Figure 4 the two sidebars acting as containers for the windows can be seen (Figure 5 displays them with a window and a collapsed window). The contents of the sidebars are fully customizable and differ per session. Windows can be collapsed and extended. Additionally, the user has the ability to select in which sidebar a specific window should be displayed.
Regardless of whether a data source is loaded in, the window sidebars are always displayed, but some windows can only be enabled if a data source is loaded. The user can collapse either (or both) sidebars.

3.1.3 Back-end

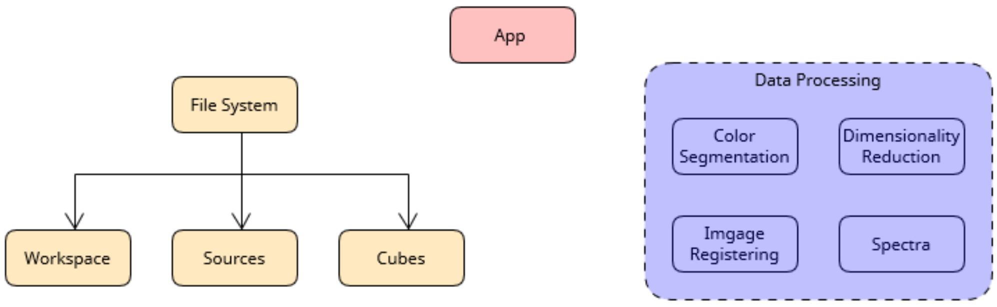

Figure 8: The back-end architecture

The back-end of the XRF Explorer 2.0 is divided into three separate components, as seen in Figure 8. The back-end was designed with modularity in mind, such that future iterations of the software can easily add more components, functionality or modify existing components. In order to achieve this, the components have precisely defined tasks, and dependencies between them have been minimized as much as possible.

App

The Flask `app` object is central in the architecture of the back-end. It serves as the point of communication with the front-end and, as such, all relevant information flows through it. Additional routes have been added to it, through which the front-end can request specific information and the `app` object will then route the request to the corresponding functionality in the back-end.

File System

The File System implements any and all basic functionality related to the files present on the server. This includes reading static files, access to the system configuration and managing the directory tree for the application. The File System does not perform any complex operations on the data.

Data Processing

All modules extracting information from the files in the data sources are bundled under the Data Processing cluster. This cluster is where the complex computations take place. These modules will mostly rely on the File System to obtain the information they need, but may also interact with each other in some rare instances, such as registering generated images to the data cubes.

3.2 Logical model description

In this section a logical model for both the front-end and back-end is given. It should be noted that these logical models (class diagrams) are an abstraction of the actual implementation intended for conveying the general structure of the software.

3.2.1 Front-end

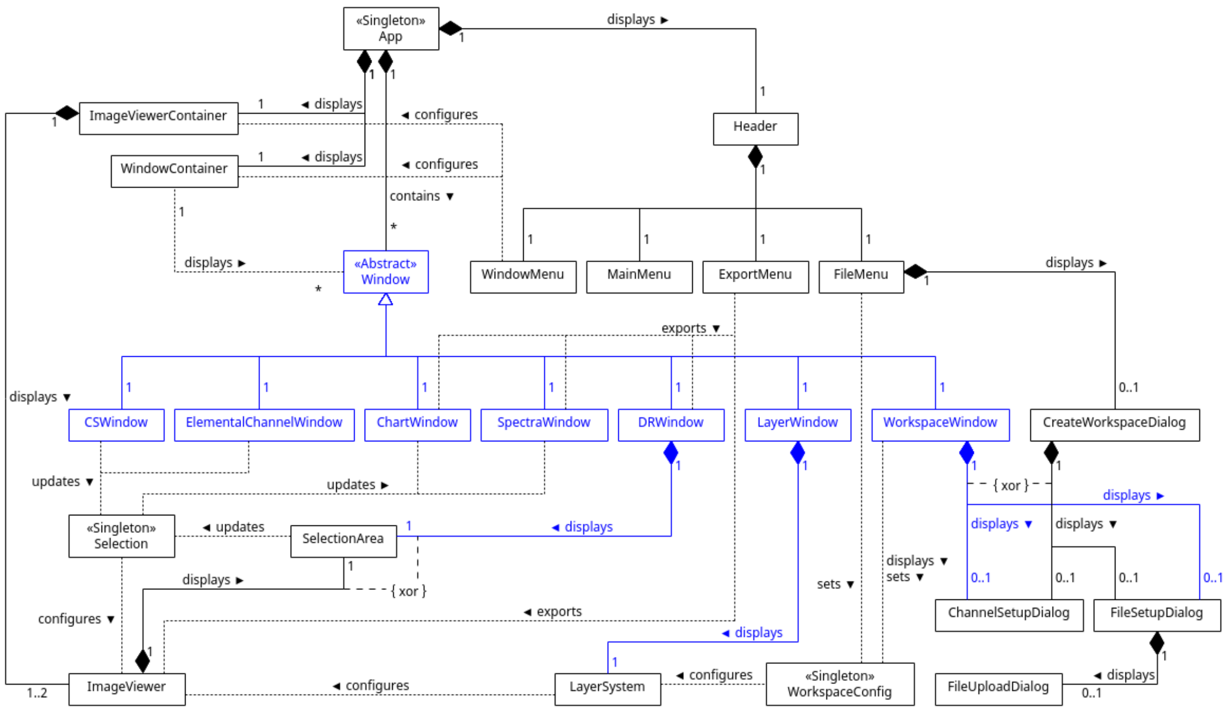

Figure 9: Logical model of the front-end

The front-end class diagram of the XRF Explorer 2.0 is given in Figure 9. Dotted lines imply a loose connection between two classes. The front-end was made using Vue.js which has reference objects. There is thus no direct connection between these classes, but to represent this communication through Vue.js reference objects, dotted lines were used. The class diagram can also be found in Appendix A in landscape format.

App

The App class has four relations in the class diagram and is the root of the front-end side of the application. The ImageViewerContainer, WindowContainer, Header and Window classes contain all of the functionality found within the software and are related and combined into a final product through the App class. Only one instance of the App class can be created and thus it is flagged as a Singleton.

- **ImageViewerContainer:** There is a composition relationship between this class and the App class as the App class holds the ImageViewerContainer and ensures it is displayed to the user. An instance of App can only have one ImageViewerContainer and each ImageViewerContainer can belong to only one App.
- **WindowContainer:** There is a composition relationship between this class and the App class as the App class holds the WindowContainer and ensures it is displayed to the user. An instance of App can only have one WindowContainer and each WindowContainer can belong to only one App.
- **Header:** There is a composition relationship between this class and the App class as the App class holds the Header and ensures it is displayed to the user. An instance of App can only have one Header and each Header can belong to only one App.
- **Window:** There is a composition relationship between this class and the App class as the App class contains the Window class. The App does not directly display the Windows but rather contains them as the process of actually displaying Windows goes through the WindowContainer. An instance of App can only have multiple Windows and each Window can belong to only one App.

The App class does not have any methods.

Header

The Header class serves as a container for the menu components. Through the Header, users can open any window and enable any functionality they may require. Thus, the Header class has connections to all menu classes for opening or enabling these features.

- **App:** The Header is displayed and contained by the App class.
- **WindowMenu, MainMenu, ExportMenu, and FileMenu:** The four menus share a composite relationship to the Header class. For each of the menus, it holds that they can only be a part of precisely one instance of the class Header and each Header class only has exactly one of each menu.

The Header class does not have any methods.

FileMenu

The FileMenu class is used for redirecting users to the CreateWorkspaceDialog and for displaying all available data sources to the user. The class mostly functions as a container class, with its only standalone functionality being an API call for getting and listing all data sources.

- **Header:** The FileMenu is displayed and contained by the Header class.
- **CreateWorkspaceDialog:** There is a composition relationship between this class and the FileMenu class as the FileMenu class holds the key to the CreateWorkspaceDialog and allows it to be displayed to the user. Each instance of CreateWorkspaceDialog belongs to one instance of FileMenu, however, it is possible for there to not be an active CreateWorkspaceDialog and thus the FileMenu either contains zero or one instance of this class.

- **WorkspaceConfig:** There is a loose relationship between this class and the FileMenu class. When selecting a workspace within the FileMenu class, the WorkspaceConfig is set by the FileMenu to the value corresponding to the selected workspace.

Name	Description
`loadWorkspace(source)`	Loads a workspace with name `source` into the back-end by making an API call for the respective back-end function.

CreateWorkspaceDialog

The CreateWorkspaceDialog class is a popup dialog, where the user can create a new data source. The user enters the name for the data source and presses ”Next” to be redirected to the next dialog (FileSetupDialog), and upon completing that dialog, is taken back to the CreateWorkspaceDialog where they can finalize the creation process by pressing ”Save”. Alternatively, the user can abort the creation by pressing the ”Abort” button which will remove all data from the back-end.

This class communicates extensively with the back-end, which will be seen in the sequence diagrams, and also shares a relationship with the ChannelSetupDialog and FileSetupDialog classes. It is the core of the data source creation process.

- **FileMenu:** The CreateWorkspaceDialog is displayed and contained by the FileMenu class.
- **ChannelSetupDialog:** There is a composition relationship between this class and the CreateWorkspace-Dialog class as the CreateWorkspaceDialog contains the ChannelSetupDialog and displays it. The ChannelSetupDialog is not always active and thus there may be zero instances of it at any given moment, it is however not possible for there to be more than one instance of it. The ChannelSetupDialog can only belong to one instance of CreateWorkspaceDialog. It should be noted, however, that the ChannelSetupDialog can also be contained within the WorkspaceWindow and be displayed through it. This cannot happen simultaneously with CreateWorkspaceDialog containing and displaying it.
- **FileSetupDialog:** There is a composition relationship between this class and the CreateWorkspace-Dialog class as the CreateWorkspaceDialog contains the FileSetupDialog and displays it. The FileSetupDialog is not always active and thus there may be zero instances of it at any given moment, it is however not possible for there to be more than one instance of it. The FileSetupDialog can only belong to one instance of CreateWorkspaceDialog. It should be noted, however, that the FileSetupDialog can also be contained within the WorkspaceWindow and be displayed through it. This cannot happen simultaneously with CreateWorkspaceDialog containing and displaying it

Name	Description
`intializeDataSource()`	Adds a data source to the list of data sources in the back-end. After that, it opens the FileSetupDialog for the user to upload files to the data source and then, if an elemental cube has been configured, opens the ChannelSetupDialog for the user to select which element channels will be used. It does not return anything.
`resetProgress()`	Resets the progress of the Workspace creation procedure. It is called after the procedure completes or if the data source is removed. It does not return anything.

Continued on next page

Name	Description
`removeDataSource()`	Removes the currently entered data source name from the back-end if a folder for it already exists. This function can only be accessed by pressing the ”abort” button in the dialog. It does not return anything.
`setupWorkspace()`	Checks if the configured workspace is valid before saving it to the back-end as a JSON file. It either returns `TRUE` for success, or `FALSE` for failure.
`updateWorkspace()`	Updates the created workspace with the new user input. First sets up the workspace with `setupWorkspace()`, initializes the elemental channels if they are not already in the workspace with `initializeChannels()`, and opens the Channel-Dialog for the user to select/deselect them. If the elemental channels are already in the workspace, it bins the spectral data with `binData()` and calls `resetWorkspace()` to terminate the set up procedure. It either returns `TRUE` for success, or `FALSE` for failure.

FileSetupDialog

The FileSetupDialog class is a popup dialog where the user can choose which files to analyze and visualize, as well as choose their names or other additional parameters regarding their storage and use. It includes a button for uploading new files (prompting the FileUploadDialog to be displayed) and a drop-down menu for selecting the type of file that should be added to the data source.

- **CreateWorkspaceDialog:** The FileSetupDialog is displayed and contained by the CreateWorkspace-Dialog. It is not always displayed, and, if it has an active relationship with WorkspaceWindow (i.e. it is currently displayed through this class), it cannot be used by the CreateWorkspaceDialog.
- **WorkspaceWindow:** The FileSetupDialog is displayed and contained by the WorkspaceWindow. It is not always displayed, and if it has an active relationship with CreateWorkspaceDialog (i.e. it is currently displayed through this class), it cannot be used by the WorkspaceWindow
- **FileUploadDialog:** There is a composition relationship between this class and the FileSetupDialog, as the class FileSetupDialog holds the FileUploadDialog and ensures it is displayed to the user. The FileUploadDialog is not always active and thus there may be zero instances of it at a given moment. It is however not possible for there to be more than one instance of it. The FileUploadDialog can only belong to one instance of FileSetupDialog.

Name	Description
`save()`	Emits the save event, prompting the containing element to save the updated setup.
`filterByExtension(filenames, extensions, empty)`	Filters a list of filenames to return only the ones with one of specific extensions. Requires `filenames`, the list of file names, `extensions`, the list of allowed extensions and `empty`, a boolean indicating whether an empty filename is added to the returned list.

Continued on next page

Name	Description
`addElementToWorkspace()`	Adds a component of the type chosen by the user to the workspace, with empty names and locations.
`removeElement(type, name)`	Removes an element from the workspace. Requires a `type`, the type of the component and `name`, its name.

FileUploadDialog

The FileUploadDialog class is a popup dialog, where the user can choose files to upload to the back-end. It has a button for opening a file explorer and a button for uploading the files.

- **FileSetupDialog:** The FileUploadDialog is displayed and contained by the FileSetupDialog class.

Name	Description
`uploadFiles()`	Uploads all the files selected by the user to the back-end by pushing them onto a queue before calling `processQueue()`. It does not return anything.
`processQueue()`	Uploads all items on the queue until it is empty by calling `uploadFile()` with each item. It does not return anything.
`uploadFile(file)`	Uploads a file to the back-end. First it divides the file into chunks before uploading them in order to the corresponding file in the data source folder of the back-end. It also updates the progress which is displayed in a progress bar. It requires `file`, the file to be uploaded. It does not return anything.
`dialogUpdate(open)`	Updates the state of the dialog by resetting its appearance if it is `open` (boolean value). Also checks that no uploads are currently being processed, and prevents the user from closing the dialog in this case. It does not return anything.

ChannelSetupDialog

The ChannelSetupDialog class is a popup dialog where the user can choose which elemental channels will be available in the workspace. It contains the list of all channels with a checkbox to select them. There is a button to select all channels at once.

- **CreateWorkspaceDialog:** Every instance of ChannelSetupDialog is a component of either the CreateWorkspaceDialog or the WorkspaceWindow component and never both.
- **WorkspaceWindow:** Every instance of ChannelSetupDialog is a component of either the CreateWorkspaceDialog or the WorkspaceWindow component and never both.

Name	Description
`toggleAllChannels()`	Toggles the selection of all channels by selecting each channel that is deselected and vice versa.

ExportMenu

The ExportMenu class is used for exporting the plots and images created within the application. As such, it is loosely connected to the windows it is able to export.

- **Header:** The ExportMenu is displayed and contained by the Header class.
- **DRWindow, SpectraWindow, ChartWindow, and ImageViewer:** The ExportMenu has a loose connection to these four classes as it is able to export the contents of these classes.

Name	Description
`exportElement(name, element)`	Exports and saves a given HTML `element` to the client as a PNG. For this, it uses an HTML Canvas function, `toBlob`, which creates a blob object of the contents of a Canvas. The function requires a `name` for the exported image. It does not return anything.

MainMenu

The MainMenu class is used to allow users to customize their work environment. From the main menu, the user can access the GitHub repository of the project, view the documentation, change the color mode/theme of the application and also reset the client.

- **Header:** The MainMenu is displayed and contained by the Header class.

The MainMenu class does not have any methods.

WindowMenu

The WindowMenu class serves as a way for users to select the windows they want to be displayed, and toggle the second main viewer. As such, the class is loosely connected to the WindowContainer and ImageViewer-Container classes.

- **Header:** The WindowMenu is displayed and contained by the Header class.
- **WindowContainer, ImageViewerContainer:** There is a loose connection between these two classes and the WindowMenu class. Through the WindowMenu class, the user can configure which windows and how many image viewers should be displayed in the WindowContainer and ImageViewerContainer classes.

Name	Description
`toggleSecondViewer()`	Toggles the visibility of the second main viewer between on and off, it does so by updating a ref variable. It does not return anything.

WindowContainer

The WindowContainer class serves as a container for the window components. In the WindowContainer, the various windows enabled by the user are visible and, therefore, it is closely related to the Window class.

- **App:** The WindowContainer is displayed and contained by the App class.
- **WindowMenu:** The contents of the WindowContainer are configured through the WindowMenu class.
- **Window:** There is an aggregation relationship between this class and the WindowContainer class, as the WindowContainer contains the Window class and displays it. Since each Window can only belong to one WindowContainer there is a multiplicity of one on this side. A WindowContainer can, however, contain any number of Windows and, as such, the multiplicity is $*$.

The WindowContainer class does not have any methods.

Window

The Window class is an abstract class from which all other window classes inherit the basic window functionality.

- **WindowContainer:** The Window is displayed and contained by the WindowContainer class.
- **CSWindow, ElementalChannelWindow, DRWindow, SpectraWindow, ChartWindow, LayerWindow, WorkspaceWindow:** There is an inheritance relationship between these classes and the Window class. The specific window classes inherit the basic window functionality from the Window class.

The Window class does not have any methods.

CSWindow

The CSWindow class is for the color segmentation functionality of the software. Through this window, the user can choose the number of clusters to compute over a specific element or the complete painting to compute them for, as well as select color clusters to be highlighted in the main viewer.

- **Window:** The CSWindow inherits basic functionality from the Window class.
- **Selection:** The CSWindow is loosely connected to the Selection class. By selecting a color from this window, the Selection singleton is updated which in turn gets communicated back to the ImageViewer, allowing the selection to be highlighted in the main viewer.

Name	Description
`fetchColors(url)`	Fetches the hexadecimal colors data through an API call to the back-end. Requires an `url` to the
`generateColors()`	Generates color clusters based on the user-set parameters, and updates the CS selection. It mak
`updateSelection()`	Called by selecting an element. Updates the CS selection by setting the index of the selected eler

ElementalChannelWindow

The ElementalChannelWindow class toggles the highlighting (and its color) of specific elements. As such, it updates the selection in a similar way to the CSWindow.

- **Window:** The ElementalChannelWindow inherits basic functionality from the Window class.
- **Selection:** The ElementalChannelWindow is loosely connected to the Selection class. By selecting an element from this window, the Selection singleton is updated which then gets communicated back to the ImageViewer, allowing the selection to be highlighted in the main viewer.

The ElementalChannelWindow class does not have any methods.

DRWindow

The DRWindow class is a window for all functionality related to dimensionality reduction. Through this window, the user is able to generate a dimensionality reduction embedding using user-supplied parameters. After the dimensionality reduction embedding has finished generating, an overlay can be chosen. Additionally, once an image has been generated in the window, the user can select an area within this image to be highlighted in the main viewer.

- **Window:** The DRWindow inherits basic functionality from the Window class.
- **ExportMenu:** The DRWindow is loosely connected to the ExportMenu, as the image generated in the DRWindow, can be exported through the ExportMenu.
- **SelectionArea:** The DRWindow contains and displays a SelectionArea class. By selecting an area of the dimensionality reduction image in this SelectionArea, the Selection class is updated which in turn gets communicated back to the ImageViewer, allowing the selection to be highlighted in the main viewer.

Name	**Description**
`fetchDRImage()`	Checks if an overlay has been specified and if an embedding is not already being generated. If everything is fine, sets the overlay type and fetches the image through an API fetch call. Makes use of a `fetchBlob` function for the canvas to visualize the image to the user. It does not return anything.
`updateEmbedding()`	Called by the create embedding button. Computes the embedding according to the given parameters. Creates an API URL for the embedding and creates the embedding. If creation and the fetching process were successful, calls the `fetchDRImage()` to load the embedding. It does not return anything.

SpectraWindow

The SpectraWindow class is a window displaying the global average of the spectrum, the average over the current selection, and the theoretical emission of a given element in the form of a chart. The user can also indicate the excitation energy element for the theoretical plot. The user can indicate, through checkboxes, which spectra they want visualized.

- **Window:** The SpectraWindow inherits basic functionality from the Window class.
- **ExportMenu:** The SpectraWindow is loosely connected to the ExportMenu as the chart generated in the SpectraWindow, can be exported through the ExportMenu.

- **Selection:** The SpectraWindow is loosely connected to the Seelection singleton. By making a selection in the main viewer, the Selection singleton is updated. This gets communicated by the Selection class to the SpectraWindow, which in turn updates the shown chart to reflect the active selection.

Name	Description
`setup()`	Sets up the SVG and axis of the graph. Awaits the `getAverageSpectrum` function and calls the `makeChart` function. It does not return anything.
`makeChart()`	Called by the global average, selection average and theoretical emission checkboxes. First clears the graph and computes the maximum y-position between the global average and selection average (iff selected) and uses this as the maximum for the y-axis. Creates the lines for the three chart types and updates the visibility according to the checkboxes. If an element has been chosen, creates peaks according to the excitation energy. It does not return anything.
`getAverageSpectrum()`	Plots the average channel spectrum over the whole painting in the chart. Makes an API fetch call for the back-end functionality for computing the global average's spectrum and calls `makeChart` upon success. It does not return anything.
`getSelectionSpectrum(selection)`	Plots the average graph of a given `selection`. Assumes that the pixels are given in the raw data coordinate system. Makes an API fetch call for the back-end functionality for computing the selection average's spectrum and calls `makeChart` upon success. It does not return anything.
`getElementSpectrum(element, excitation)`	Plots the theoretical spectrum and peaks of an `element` with a given `excitation` energy. Makes an API fetch call for the back-end functionality for computing the theoretical emission of a given `element` with a given `excitation` energy and alls `makeChart` upon success. It does not return anything.

ChartWindow

The ChartWindow class is a window displaying the charts of the abundance per element. Within the window, the user can choose whether they want to see the global average as a bar chart, the average over the current selection as a line chart, or both simultaneously.

- **Window:** The ChartWindow inherits basic functionality from the Window class.
- **ExportMenu:** The ChartWindow is loosely connected to the ExportMenu as the chart generated in the ChartWindow can be exported through the ExportMenu.
- **Selection:** The ChartWindow is loosely connected to the Selection singleton. By making a selection in the main viewer, the Selection singleton is updated. This gets communicated by the Selection class to the ChartWindow, which in turn updates the shown chart to reflect the active selection. Additionally, elements selected in the ElementalChannelWindow are also communicated to the ChartWindow through the Selection singleton, these elements are then highlighted in the bar chart.

Name	Description
`fetchAverages(url, selectionRequest, selection)`	Fetches the average elemental data for each of the elements, and stores it in an array. Makes an API fetch call to a given `url` which provides the elemental data; given that the boolean `selectionRequest` is `TRUE`. Requires a `selection` made in the main viewer. Returns `TRUE` if the averages were fetched correctly and `FALSE` otherwise.
`setupChart(data)`	Sets up the chart's SVG container and adds the axes. Requires `data`, the elemental data array which should be displayed on the chart. Adjusts the y-axis according to the maximum y value of the given `data`. It does not return anything.
`updateLineChart(data)`	Given an elemental `data` array which should be displayed on the chart. Adds the line chart to the SVG container with updated `data`. It does not return anything.
`updateBarChart(data)`	Given an elemental `data` array which should be displayed on the chart. Adds the bar chart to the SVG container with updated `data`. Sets the color of the bar according to the color selected in the ElementalChannelWindow. It does not return anything.

LayerWindow

The LayerWindow class serves as a container for the LayerSystem where the user can make changes to specific layers and update the layer hierarchy.

- **Window:** The LayerWindow inherits basic functionality from the Window class.
- **LayerSystem:** There is a composition relationship between this class and the LayerWindow class, as the LayerWindow contains and displays the LayerSystem class. An instance of LayerWindow can only have one LayerSystem and each LayerSystem can belong to only one LayerWindow.

The LayerWindow class does not have any methods.

LayerSystem

The LayerSystem class contains all functionality related to the layer stack. Within this class, users are able to use filters, update the layer hierarchy and toggle the visibility of layers. Additionally, the user can choose which layer should be visible in the lens.

- **LayerWindow:** The LayerSystem is displayed and contained by the LayerWindow class.
- **WorkspaceConfig:** As WorkspaceConfig singleton describes what types of data are available in the client, it configures the contents of the LayerSystem. This is a loose connection.
- **ImageViewer:** The LayerSystem configures the contents of the ImageViewer, since it holds the information about how each layer should be displayed, this is a loose connection. By updating any of the settings related to a specific layer, the ImageViewer is reconfigured. The ImageViewer class displays the contents and configurations made in the LayerSystem. It does this indirectly, as unlike the LayerWindow, the ImageViewer does not necessarily display the LayerSystem itself, but rather the state of the layers.

Name	Description
`checkedOutsideLens(group)`	Updates the visibility of the layer `group` outside of the lens. This function is triggered by checking the "Visible inside lens" checkbox. It does not return anything.

WorkspaceWindow

The WorkspaceWindow class contains an overview of the current WorkspaceConfig and gives the user a place to edit the active workspace by navigating them to the FileSetupDialog and ChannelSetupDialog class.

- **WorkspaceConfig:** There is a loose connection between the WorkspaceWindow class and the WorkspaceConfig class. Through the WorkspaceWindow class, an overview of the current WorkspaceConfig is given. Additionally, through the same FileSetupDialog and ChannelSetupDialog used by the CreateWorkspaceDialog in creating the workspace, the WorkspaceWindow allows the user to update and change the WorkspaceConfig. Since ChannelSetupDialog and FileSetupDialog work on a local copy of the WorkspaceConfig, instead of directly modifying the WorkspaceConfig, this relation is not drawn from the dialogs where the changes are visually being made by the user.
- **ChannelSetupDialog:** There is a composition relationship between this class and the WorkspaceWindow class, as the WorkspaceWindow contains and displays the ChannelSetupDialog The ChannelSetupDialog is not always active and thus there may be zero instances of it at any given moment. It is however not possible for there to be more than one instance of it. The ChannelSetupDialog can only belong to one instance of WorkspaceWindow. It should be noted, however, that the ChannelSetupDialog can also be contained within the WorkspaceWindow and be displayed through it. This can not happen simultaneously with WorkspaceWindow containing and displaying it.
- **FileSetupDialog:** There is a composition relationship between this class and the WorkspaceWindow class as the WorkspaceWindow contains and displays the FileSetupDialog. The FileSetupDialog is not always active and thus there may be zero instances of it at any given moment. It is however not possible for there to be more than one instance of it. The FileSetupDialog can only belong to one instance of WorkspaceWindow. It should be noted, however, that the FileSetupDialog can also be contained within the WorkspaceWindow and be displayed through it. This cannot happen simultaneously with WorkspaceWindow containing and displaying it.

Name	Description
`updateWorkspace()`	Updates the workspace persistently. Sends a POST API request to the back-end for updating the workspace and updates the app state and display if this call was successful. It does not return anything.

WorkspaceConfig

The WorkspaceConfig class contains the details of the active workspace's configuration. It stores the following information about the workspace: the name, base image, contextual images, spectral cubes, elemental cubes, elemental channels, and parameters used to read the spectral data.

- **FileMenu:** Through the FileMenu the user can choose and set the current workspace.
- **WorkspaceWindow:** The details of the WorkspaceConfig class are displayed in the WorkspaceWindow and may be set through this window.
- **LayerSystem:** The WorkspaceConfig determines which images and layers are available, configuring the LayerSystem.

The WorkspaceConfig does not have any methods.

ImageViewerContainer

The ImageViewerContainer class serves as a container for the image viewers. This class stores the logic for whether one or two image viewers should be displayed simultaneously, for it reads a reference variable.

- **App:** The ImageViewerContainer is displayed and contained by the App class.
- **WindowMenu:** The content of the ImageViewerContainer is configured through the WindowMenu class.
- **ImageViewer:** There is a composition relationship between this class and the ImageViewerContainer class, as the ImageViewerContainer class contains and displays the ImageViewer class. Each ImageViewer can only correspond to one instance of the ImageViewerContainer class. Since there is the possibility for two ImageViewers to be enabled simultaneously, the ImageViewerContainer class can contain two ImageViewers (and must always contain at least one).

The ImageViewerContainer does not have any methods.

ImageViewer

The ImageViewer class allows the actual painting (scene) to be visualized to the user. Within this class, a toolbar is visible for panning and creating a selection. Most importantly, in the ImageViewer class, the scene is rendered to the user allowing them to see the painting and all configurations which have been made through the earlier window classes.

- **ImageViewerContainer:** The ImageViewer is displayed and contained by the ImageViewerContainer class.
- **SelectionArea:** There is a composition relationship between this class and the ImageViewer class, as the ImageViewer contains and displays the SelectionArea class. Every instance of SelectionArea must be a component of the ImageViewer or DRWindow, but never both.
- **Selection:** The ImageViewer and Selection singleton are loosely connected. When the active selection gets changed, this is communicated to the ImageViewer which in turn updates and rerenders the visualization.

Name	Description
`setup()`	Sets up the very basic scene in THREE for rendering and calls the `render` function. It does not return anything.
`render()`	Calculates the viewport parameters and renders a frame in the renderer. The function makes use of the Web API `requestAnimationFrame` method, which informs the browser that the user wants to perform an animation. It does not return anything.
`clearSelection()`	Cancels the selection made in the image viewer.
`onMouseDown()`	Event handler for the onMouseDown event on the GLCanvas, enables ”panning mode” if the selection tool is inactive. If the user right-clicks, it updates the `lensLocked`. It does not return anything.
`onMouseUp()`	Event handler for the onMouseUp event on the GLCanvas, disables ”panning mode”. It does not return anything.

Continued on next page

Name	**Description**
`onMouseMove(event)`	Event handler for the onMouseMove event on the GLCanvas. If the ”panning mode” is enabled, modifies the viewport given an `event` (contains the movement of the mouse). Ensures correct mapping of the coordinate system by reversing the y-axis such that (0,0) is at the top left. If `lensLocked` is false, the shader position of the lens is updated. It does not return anything.
`onWheel(event)`	Event handler for the onWheel event on the GLCanvas. Modifies the viewport to allow zooming in and out on the painting, with the zoom being clamped at a reasonable range to prevent infinite zooming. It requires an `event` containing the amount that was scrolled. It does not return anything.
`loadLayer(layer, interpolated)`	Creates a layer in the image viewer and adds the given image to it. First checks the boolean `interpolated` to see whether interpolation should be disabled or not. Then creates a THREE-shape with the dimensions of the painting (texture) and adds the texture to allow it being used in the shaders. Afterwards, it creates a material to render the texture onto the created shape and sets the initial order of the layers. A vertex shader handles the movement of the texture for registering and the viewport, and a fragment shader handles sampling colors from the texture. It does not return anything.
`createDataTexture(data, width, height)`	Creates a data texture of specified size and format wherein every pixel contains 4 bytes of data. Requires a `data` array and the `width` and `height` of the data texture. It returns the data texture as a THREE DataTexture object.

SelectionArea

The SelectionArea class contains the details of the currently active selection. It has functions for the creation of both polygon and rectangle selections and makes use of a special type, `SelectionAreaType`. It has an inactive mode when no selection tool is selected and two active modes: one for the polygon selection tool and another one for the rectangle selection tool.

- **CSWindow, ElementalChannelWindow, and DRWindow:** From these three classes, the user can select elements or areas to be highlighted in the main viewer. This updates the SelectionArea class.
- **ImageViewer:** The user is able to make a selection in the ImageViewer class which requires the SelectionArea and also updates it. Additionally, the SelectionArea configures the ImageViewer by informing it of selections made in different windows.
- **SpectraWindow, ChartWindow:** Making a selection in the ImageViewer updates the SelectionArea. This updated selection can be displayed in the SpectraWindow and ChartWindow making it such that the SelectionArea updates the SpectraWindow and ChartWindow classes.

Name	Description
`onClick(event)`	Handles a click `event` for polygon selection. First checks whether a selection is already started, and ends it if the clicked point is close to the starting position. Otherwise, it adds a new point to the selection. The polygon selection tool needs to be activated for this function to be executed. It does not return anything.
`onMouseDown(event)`	Starts a rectangle selection given a mouse down `event`. The rectangle selection tool needs to be activated for this function to be executed. It does not return anything.
`onMouseUp()`	Ends a rectangle selection. The rectangle selection tool needs to be activated for this function to be executed. It does not return anything.

Selection

The Selection class contains the details of the currently active selection. The currently active selection consists of four different selections, the selection made in the main viewer, the selection made in the DRWindow, the selection made in the CSWindow and the selection made in the ElementalChannelWindow. As the currently active is part of the global application state, the Selection class is a singleton.

- **CSWindow, ElementalChannelWindow** From these two classes, the user can select elements or color clusters to be highlighted in the main viewer. When such a selection is made in these windows, the window updates the Selection singleton.
- **SelectionArea:** The user is able to make an area selection in the ImageViewer and DRWindow classes using the respective SelectionArea instances contained in those classes. When such a selection is made, SelectionArea updates the Selection singleton.
- **SpectraWindow, ChartWindow:** When the main viewer selection is updated in the Selection singleton, this gets communicated to the SpectraWindow and ChartWindow. In turn, the SpectraWindow and ChartWindow update and redraw the charts to reflect the active selection.
- **ImageViewer:** When the DRWindow selection, ElementalChannelWindow selection or CSWindow selection is updated in the Selection singleton, this gets communicated to the ImageViewer. In turn, the ImageViewer instances update and rerender to reflect the active selection.

The Selection class has no methods.

3.2.2 Back-end

The back-end class diagram of the XRF Explorer 2.0 is given in Figure 10. Here, the three main components of the back-end are visible: the File System in purple, the Flask API in red, and the data processing components in blue. The relationships between the data processing components and the File System are shown in black.

The back-end was built using the Flask framework, which communicates to the front-end using the `app` object and the API endpoints as defined in the routes module. These will usually request some computation on the uploaded data from a data processing component, but may also simply request information on the file system itself.

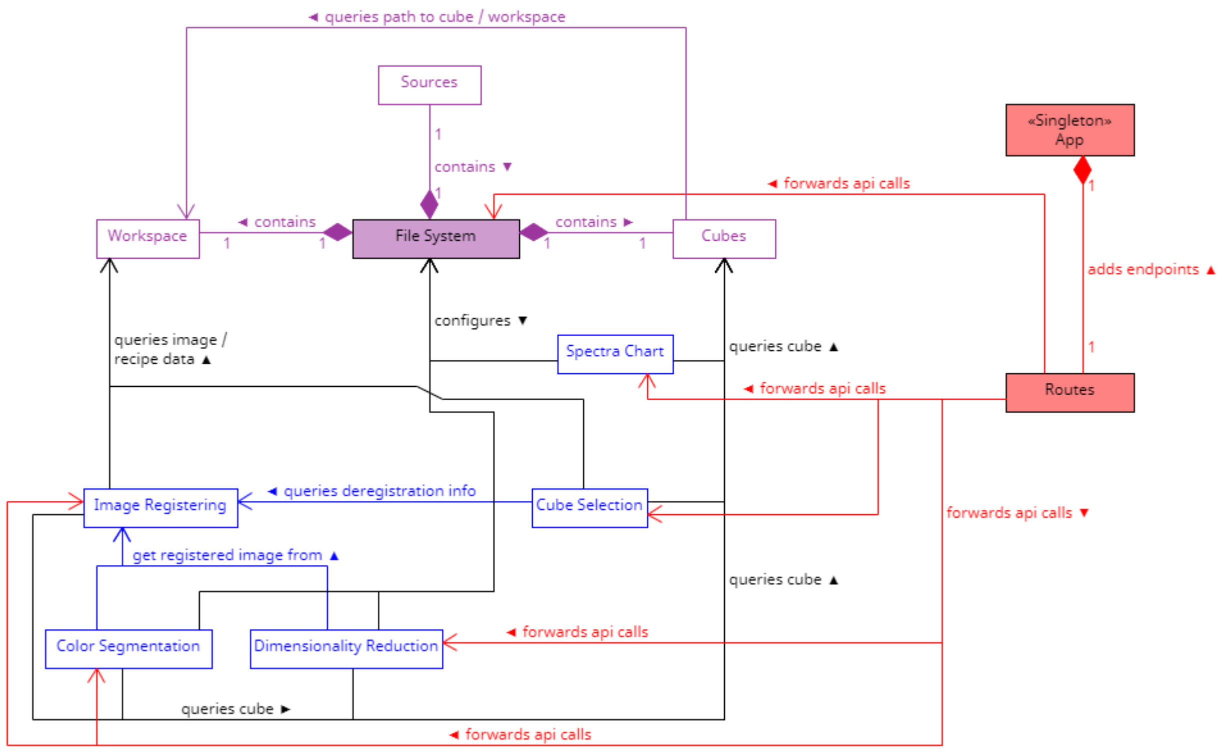

Figure 10: Logical model of the back-end

Routes

The routes module defines all the API endpoint extensions to be appended to the Flask `app` object. It routes API calls from the front-end to the relevant functionality in the back-end and returns the appropriate response. The following table defines all API endpoints the routes module adds to the `app` object.

Route	Method(s)	Parameters	Description
`/api`	GET		Returns a list of all API endpoints.
`/api/data_sources`	GET		Returns a list of all available data sources stored in the data folder of the remote server.
`/api/<data_source>/workspace`	GET & POST	`data_source`: The name of the data source to get the workspace content for.	Returns the workspace content for the specified data source or writes to it if a POST request is made.
`/api/<data_source>/files`	GET	`data_source`: The name of the data source to get the files for.	Returns a list of all available files for a data source.

Continued on next page

Route	Method(s)	Parameters	Description
`/api/<data_source>/create`	POST	`data_source`: The name of the data source to create.	Creates a directory in the data folder of the remote server for a new data source.
`/api/<data_source>/remove`	POST	`data_source`: The name of the data source to be aborted.	Removes workspace.json from a data source.
`/api/<data_source>/delete`	DELETE	`data_source`: The data source to be deleted.	Removes all files from data source.
`/api/<data_source>/upload/<file_name>/<start>`	POST	`data_source`: The name of the data source to upload the chunk to. `file_name`: The name of the file to upload the chunk to. `start`: The start index of the chunk in the specified file.	Uploads a chunk of bytes to a file in specified data source.
`/api/<data_source>/data/convert`	GET	`data_source`: The name of the data source containing the elemental data cubes to be converted.	Converts all elemental data cubes of a data source to .dms format.
`/api/<data_source>/bin_raw`	POST	`data_source`: The data source containing the raw data to bin.	Bins the raw data files channels to compress the file.
`/api/<data_source>/get_offset`	GET	`data_source`: The name of the data source containing the raw data.	Returns the depth offset energy of the raw data, that is the energy level of channel 0.
`/api/<data_source>/element_averages`	GET & POST	`data_source`: The name of the data source to get the element averages from.	Returns the names and averages of the elements present in the painting.
`/api/<data_source>/element_averages_selection`	POST	`data_source`: The name of the data source to get the element averages from.	Returns the names and averages of the elements present in a selection of the painting.
`/api/<data_source>/data/elements/names`	GET	`data_source`: The name of the data source to get the element names from.	Returns the short names of the elements stored in the elemental data cube.

Continued on next page

Route	Method(s)	Parameters	Description
`/api/<data_source>/dr/embedding/<element>/<threshold>`	GET	`data_source`: The name of the data source to generate the embedding from. `element`: The element to generate the embedding for. `threshold`: The threshold from which a pixel is selected.	Generates the dimensionality reduction embedding of an element, given a threshold.
`/api/<data_source>/dr/overlay/<overlay_type>`	GET	`data_source`: The name of the data source to get the overlay from. `overlay_type`: The overlay type: Images are prefixed with `contextual_` and element by `elemental_`	Generates the dimensionality reduction overlay with a given type.
`/api/<data_source>/dr/embedding/mapping`	GET	`data_source`: The name of the data source to get the overlay from.	Creates the image for a selection that decodes to which points in the embedding the pixels of the elemental data cube are mapped. Uses the current embedding and indices for the given data source to create the image.
`/api/<data_source>/image/<name>`	GET	`data_source`: The name of the data source to get the image from. `name`: The name of the image in workspace.json	Returns the requested contextual image.
`/api/<data_source>/image/<name>/size`	GET	`data_source`: The name of the data source to get the image from. `name`: The name of the image in workspace.json	Returns the size of the requested contextual image.
`/api/<data_source>/image/<name>/recipe`	GET	`data_source`: The name of the data source to get the image recipe from. `name`: The name of the image in workspace.json	Gets the registering recipe of the requested contextual image.
`/api/<data_source>/data/size`	GET	`data_source`: The name of the data source to get the size from.	Returns the size of the data cubes.

Continued on next page

Route	Method(s)	Parameters	Description
`/api/<data_source>/data/recipe`	GET	`data_source`: The name of the data source to get the recipe from.	Returns the registering recipe for the data cubes.
`/api/<data_source>/data/elements/map/<channel>`	GET	`data_source`: The name of the data source to get the elemental map from. `channel`: The channel to get the map from.	Returns the requested elemental map.
`/api/<data_source>/get_average_data`	GET	`data_source`: The name of the data source to get the raw data from.	Computes the average of the raw data for each bin of channels in the range [low, high] on the whole painting.
`/api/<data_source>/get_element_spectrum/<element>/<excitation>`	GET	`data_source`: The name of the data source to get the spectrum boundaries and the bin size from. `element`: The chemical element to get the theoretical spectra of. `excitation`: The excitation energy of the element.	Computes the theoretical spectrum in the channel range [low, high] for an element with a bin size, as well as the element's peak energies and intensity.
`/api/<data_source>/get_selection_spectrum`	POST	`data_source`: The name of the data source to get the average spectrum from.	Returns the average spectrum of the selected pixels of a selection.
`/api/<data_source>/cs/clusters/<elem>/<k>/<elem_threshold>`	GET	`data_source`: The name of the data source to get the clusters from. `elem`: The index of the selected element (0 if we want the whole painting, $channel + 1$ if we want a specific element). `k`: The number of color clusters to compute. `elem_threshold`: The elemental threshold.	Returns the colors corresponding to the image-wide/element-wise color clusters and caches them as well as the corresponding bitmask.

Continued on next page

Route	Method(s)	Parameters	Description
`/api/<data_source>/cs/bitmask/<elem>/<k>/<elem_threshold>`	GET	`data_source`: The name of the data source to get the bitmask from. `elem`: The index of the selected element (0 if we want the whole painting, $channel + 1$ if we want a specific element). `k`: The number of color clusters to compute. `elem_threshold`: The elemental threshold.	Returns the PNG bitmask for the color clusters over the whole painting/selected element.

File System

The File System module contains a series of sub-modules implementing basic functionality related to the various files stored on the server. This includes reading configuration or data files, amongst others. Since the File System module performs only basic functions, it is a self-sufficient entity that exists independently from the rest of the system.

- **Cubes, Sources, Workspace:** The three sub-modules share a composite relationship to the File System module, as they rely on functionality implemented by the File System module itself. Each module is only ever initialized exactly once.

Name	Description
`set_config(path)`	Parses and sets the configuration read from the given YML file globally to be accessed by the back-end.
`get_config()`	Returns the set configuration for the back-end as a dictionary.
`get_path_to_generated_folder(data_source)`	Returns the path to the `generated` folder (the folder where generated files are stored) for a given data source. If the folder does not exist, it will be created.

Cubes

The Cubes module handles basic tasks related to the elemental and spectral cubes, which provide information about the chemical composition of the painting. An important distinction between this module and similar processing modules is that this module is only for reading basic information from the cubes or creating new ones.

- **File System:** The Cubes module is part of the File System and requires basic functionality implemented by the latter.
- **Workspace:** The Cubes module does not know which data source is currently loaded. This information is extracted from the Workspace module.

Name	Description
`convert_elemental_cube_to_dms(data_source, cube_name)`	Converts an elemental data cube to the preferred .dms format. It also updates the workspace accordingly and removes the old elemental data cube.
`get_element_names(path)`	Returns a list of the names of the elements present in the painting.
`get_elemental_data_cube(data_source)`	Returns a 3-dimensional numpy array containing the elemental data cube from a given data source. The first dimension is the channel (corresponding to the element), and the last two are x, y coordinates.
`get_elemental_map(element, path)`	Returns the elemental map of element index at the given path. Returns a 2-dimensional numpy array containing the elemental data cube, where the dimensions are the x, y coordinates.
`get_raw_data(data_source, level)`	Parses the raw data cube of a data source as a 3-dimensional numpy array.
`get_spectra_params(data_source)`	Returns the spectrum parameters (low/high boundaries and bin size) of a data source.
`parse_rpl(path)`	Parses the .rpl file of a data source as a dictionary, containing the following information: width, height, depth, offset, data length, data type, byte order, record by.
`get_elemental_datacube_dimensions(data_source)`	Returns a 4-tuple of the dimensions of the elemental data and the header size (in bytes). Tuple is as follows: `(width, height, channels, header size)`.

Sources

The Sources module is responsible for handling the different data sources from a high level perspective, without accessing or manipulating the individual files. It provides the front-end with information about the different data sources available for use and the files they contain.

- **File System:** The Sources module is part of the File System and requires basic functionality implemented by it.

Name	Description
`get_data_sources_names()`	Returns a list of all available data sources stored in the data folder on the remote server as specified in the project's configuration.
`get_data_source_files(data_source)`	Returns a list of all the files stored in the folder belonging to a given data source. It does not look at files in sub-directories.

Workspace

The Workspace module handles the extraction of information from a specific data source, as well as maintaining the state of each data source. It provides a level of abstraction away from individual data sources for the rest of the components.

- **File System:** The Workspace module is part of the File System and requires basic functionality implemented by the latter.

Name	Description
`get_base_image_path(data_source_folder_name)`	Returns the path to the RGB image of the given data source.
`get_base_image_name(data_source_folder_name)`	Returns the name of the RGB image of a given data source.
`get_contextual_image_path(data_source, name)`	Returns the path of the requested contextual image.
`get_contextual_image_size(image_path)`	Returns the size of an image in pixel dimensions.
`get_path_to_base_image(data_source)`	Returns the path to the base image of the given data source.
`get_elemental_cube_path(data_source)`	Returns the path to the elemental data cube of a data source.
`get_elemental_cube_path_from_name(data_source, cube_name)`	Returns the path to the elemental data cube of the given cube and data source.
`get_raw_rpl_paths(data_source)`	Returns the paths to the raw data file (.raw) and the .rpl file of a given data source.
`get_raw_rpl_names(data_source)`	Returns the name of the raw data file (.raw) and the .rpl file of a given data source.
`get_workspace_dict(data_source_folder_name)`	Returns the `workspace.json` file of the specified data source in dictionary format.
`get_path_to_workspace(datasource)`	Returns the path to the `workspace.json` file for a given data source.
`update_workspace(datasource, new_workspace)`	Updates the `workspace.json` file for a given data source with the given values.

Image Registration

The Image Registration module aligns images with other images or (the image representation of) data cubes, ensuring that corresponding points across the images match precisely in the same coordinate system.

- **Cubes:** The Cubes module is used for extracting the dimensions of the elemental or spectral cubes when registering to them.
- **Workspace:** Information necessary for registration is extracted from the data source using the workspace module.

Name	Description
`get_image_registered_to_data_cube(data_source, image_name)`	Aligns an image to the dimensions of a data cube using series of transformations - resizing, padding, and perspective adjustment. The function requires two arguments: the name of the data source folder, which contains the data cube, the data cube recipe, and the reference image; and the name of the image to be registered. It returns the matrix of the image as numpy array if registration is successful and None otherwise.
`inverse_register_image(image, new_width, new_height, points_source, points_destination)`	Inverses the registration of a given image by reversing the perspective transformation using the source and destination points in reversed order. It then scales and pads the image to match the specified width and height. The function returns the matrix of the inverse registered image.
`register_image(image, new_width, new_height, points_source, points_destination)`	Aligns an image to match the specified width and height using a series of transformations. The function resizes the image while maintaining the aspect ratio, pads it to the desired dimensions, and then applies a perspective transformation using the provided source and destination points. It returns the matrix of the registered image.
`load_points(path_points_csv_file)`	Loads the control points for a perspective transformations from a .csv file. The format of the .csv file must be as generated by the butterfly registrator [3]. The function returns a tuple src_points, dst_points.
`apply_perspective_transformation( image_to_transform, points_src, points_dest)`	Applies a perspective transformation on a given image based on source and destination points. The function returns the matrix of the transformed image.
`pad_image_to_match_size(image_to_pad, image_reference_height, image_reference_width)`	Pads the given image to match the given dimension by either adding or removing rows/columns. The function returns the matrix of the padded image.
`resize_image_fit_aspect_ratio(image_resize, image_reference_height, image_reference_width)`	Resizes the given image to match the aspect ratio calculated by the given reference width and height. The resizing is done by scaling and padding. The function returns matrix of the resized image.

Cube Selection

The Cube Selection module handles the precise extraction of pixels in the high resolution images, as well as the chemical data contained therein, from a selection made in the front-end (of a given type, i.e. Rectangle or Polygon).

- **Workspace:** The Workspace module provides the paths to the necessary cubes, images and recipes.
- **Cubes:** The dimensions of the cubes are extracted from the Cubes module.
- **Image Registering:** All information about the contextual images required to fit the cube to the images is obtained from the Image Registering module, such that the registering can be reversed.

Name	Description
`compute_selection_mask(selection_type, selection, cube_width, cube_height)`	Given a list of coordinates (representing the edges of the polygon selection), selection type, and the cube dimensions, the function computes the selection mask of the data cube image.
`get_selection(data_source_folder, selection_coords, selection_type, cube_type)`	Given the name of the data source folder, a list of coordinates (representing the edges of the polygon selection), type of selection and type of cube, the function computes the selection maskof the data cube image.

Color Segmentation

As the name suggests, the Color Segmentation module is responsible for handling any API requests regarding the color segmentation functionality, such as computing the clustering of colors for the whole painting or by element.

- **File System:** The File System provides the back-end configuration, as well as the path to the folder containing generated files, where the generated clusters are stored for better performance and user experience.
- **Cubes:** The elemental cube data is obtained from the Cubes module.
- **Image Registering:** The Image Registering module provides the functionality to register the RGB image, from which the colors are extracted, to the elemental cube's dimensions.

Name	Description
`combine_bitmasks(bitmasks)`	Merges an array of bitmasks into a single bitmask, by setting the Green value of each pixel to store the index of the corresponding bitmask.
`get_clusters_using_k_means(data_source, image_name, k, nr_of_attempts)`	Extracts the color clusters of the RGB image using the k-means clustering method, where the number of clusters `k` is specified.
`get_elemental_clusters_using_k_means(data_source, image_name, elemental_channel, elem_threshold, k, nr_of_attempts)`	Extracts the color clusters of the RGB image per elemental channel using the k-means clustering method, where the number of clusters `k` is specified.
`save_bitmask_as_png(bitmask, full_path)`	Saves the given bitmask as a PNG with the given name at the given path.

Dimensionality Reduction

The Dimensionality Reduction module is tasked with performing all the computations for the dimensionality reduction window in the front-end. Since these computations are quite complex and take a significant amount of time to compute, the results are always stored locally for faster access when the same generated data is reused.

- **Image Registering:** The Image Registering module provides the functionality to register a generated image to the elemental cube's dimensions.
- **File System:** The File System provides the back-end configuration, as well as the path to the folder containing generated files, where the generated data and images are stored for better performance and user experience.

- **Cubes:** The Cubes module provides the data contained in the elemental cube, as well as functionality to normalize the cube to grayscale.

Name	Description
`generate_embedding(data_source, element, threshold, new_umap_parameters)`	Generates the embedding (i.e. a lower dimensional representation of the data) of the elemental data cube using the dimensionality reduction method "UMAP". The embedding is generated for a specified element and uses a specified threshold to filter the data cube by. It also generates an image encoding a mapping from the embedding back to the elemental data cube. The results is stored in the folder specified in the project configuration.
`get_image_of_indices_to_embedding(data_source)`	Returns the path to the image that maps the indices of the elemental data cube to the embedding.
`create_embedding_image(data_source, overlay_type)`	Creates the embedding image from the embedding, with the specified overlay. Returns the path to the created embedding image if successful, otherwise an empty string.

Spectra

The Spectra module uses the spectral cubes to perform the computations necessary to display the average fluorescence spectrum requested by the SpectraWindow.

- **File System:** The File System provides the back-end configuration for the Spectra module.
- **Cubes:** The Cubes module provides the data contained in the spectral file for the Spectra module to perform the requested calculations.

Name	Description
`get_average_global(data)`	Computes the average of the raw data for each bin on the whole painting. Returns a list where the index is the channel number and the value is the average global intensity of that channel.
`get_average_selection(data_source, mask)`	Computes the average of the raw data for each bin on the selected pixels. Returns a list where the index is the channel number and the value is the average intensity of that channel within the selection.
`get_theoretical_data(element, excitation_energy_kev, low, high, bin_size)`	Returns a list with first element being a list of dictionaries representing the spectra points, second being a list of dictionaries representing the peaks of a specified element.

3.3 System dynamics

3.3.1 Upload a data source

To upload a data source, the user clicks ”File” and then ”New project”. In the pop-up, the user has to choose a name for the data source, and then click on ”Next”. A new pop-up will appear, where the user clicks ”Upload files” to upload all relevant files. Then, the user chooses the data source elements by mapping them to the uploaded files. Finally, the user clicks the ”Save” button”, and the system creates the data source.

Goal	To upload a data source
Preconditions	None
Postconditions	A new data source is created
User	All
Priority	Must have

1

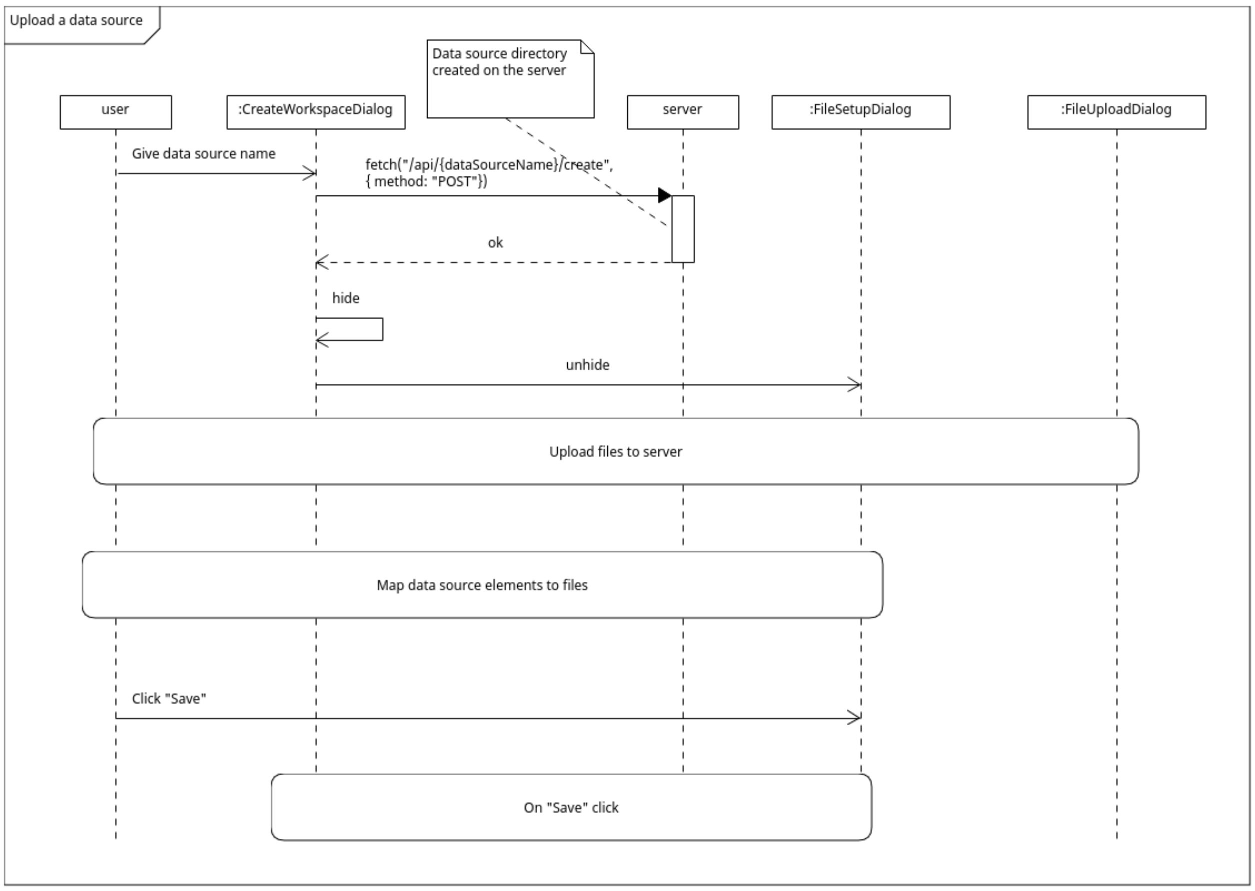

Figure 11: A. Upload a data source

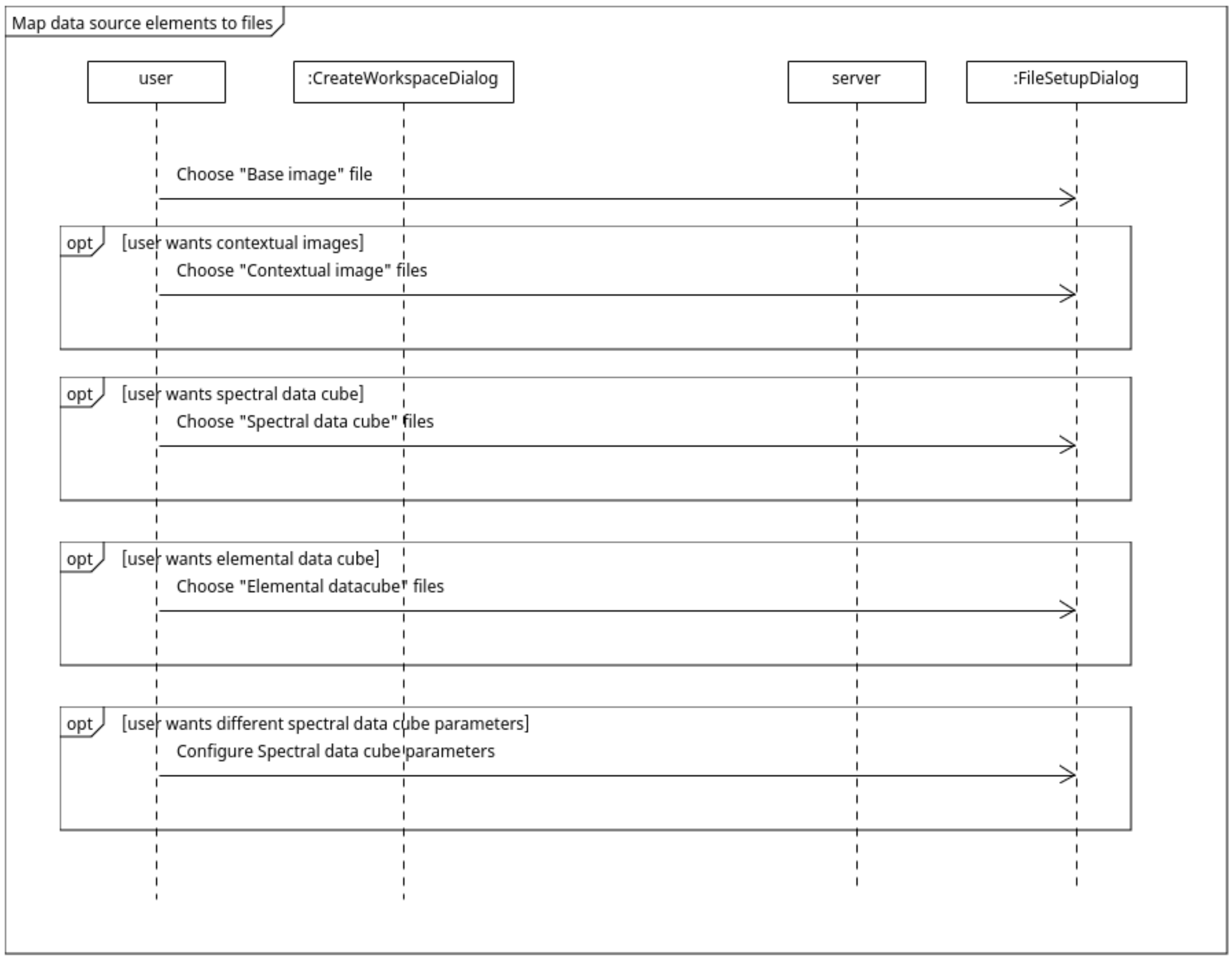
Map data source elements to files
user
:CreateWorkspaceDialog
server
:FileSetupDialog
Choose "Base image" file
opt
[user wants contextual images]
Choose "Contextual image" files
opt
[user wants spectral data cube]
Choose "Spectral data cube" files
opt
[user wants elemental data cube]
Choose "Elemental datacube" files
opt
[user wants different spectral data cube parameters]
Configure Spectral data cube parameters

Figure 12: B. Upload a data source

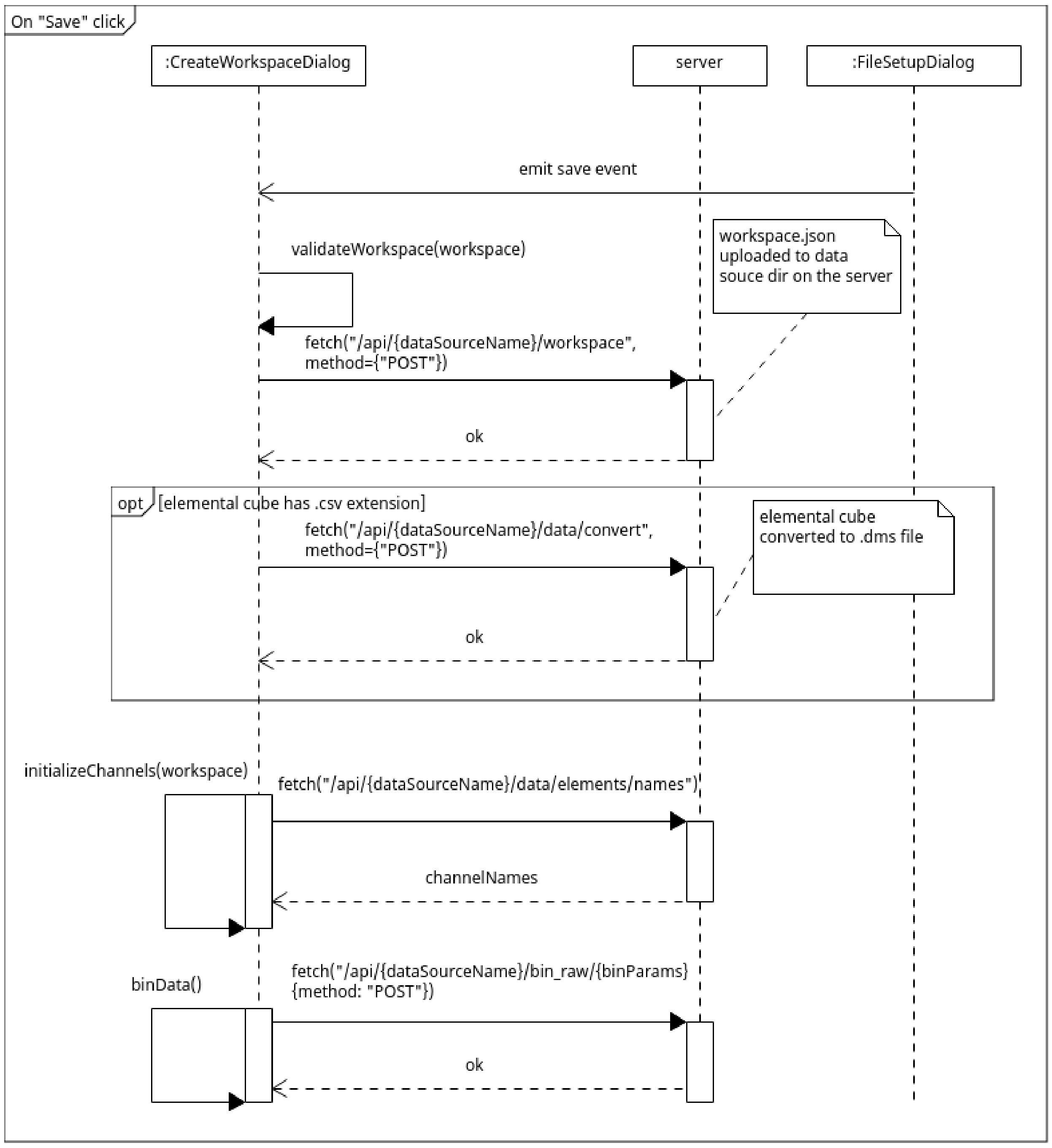
On "Save" click
:CreateWorkspaceDialog
server
:FileSetupDialog
emit save event
validateWorkspace(workspace)
workspace.json uploaded to data souce dir on the server
fetch("/api/{dataSourceName}/workspace", method={"POST"})
ok
opt [elemental cube has .csv extension]
fetch("/api/{dataSourceName}/data/convert", method={"POST"})
elemental cube converted to .dms file
ok
initializeChannels(workspace)
fetch("/api/{dataSourceName}/data/elements/names")
channelNames
binData()
fetch("/api/{dataSourceName}/bin_raw/{binParams} {method: "POST"})
ok

Figure 13: C. Upload a data source

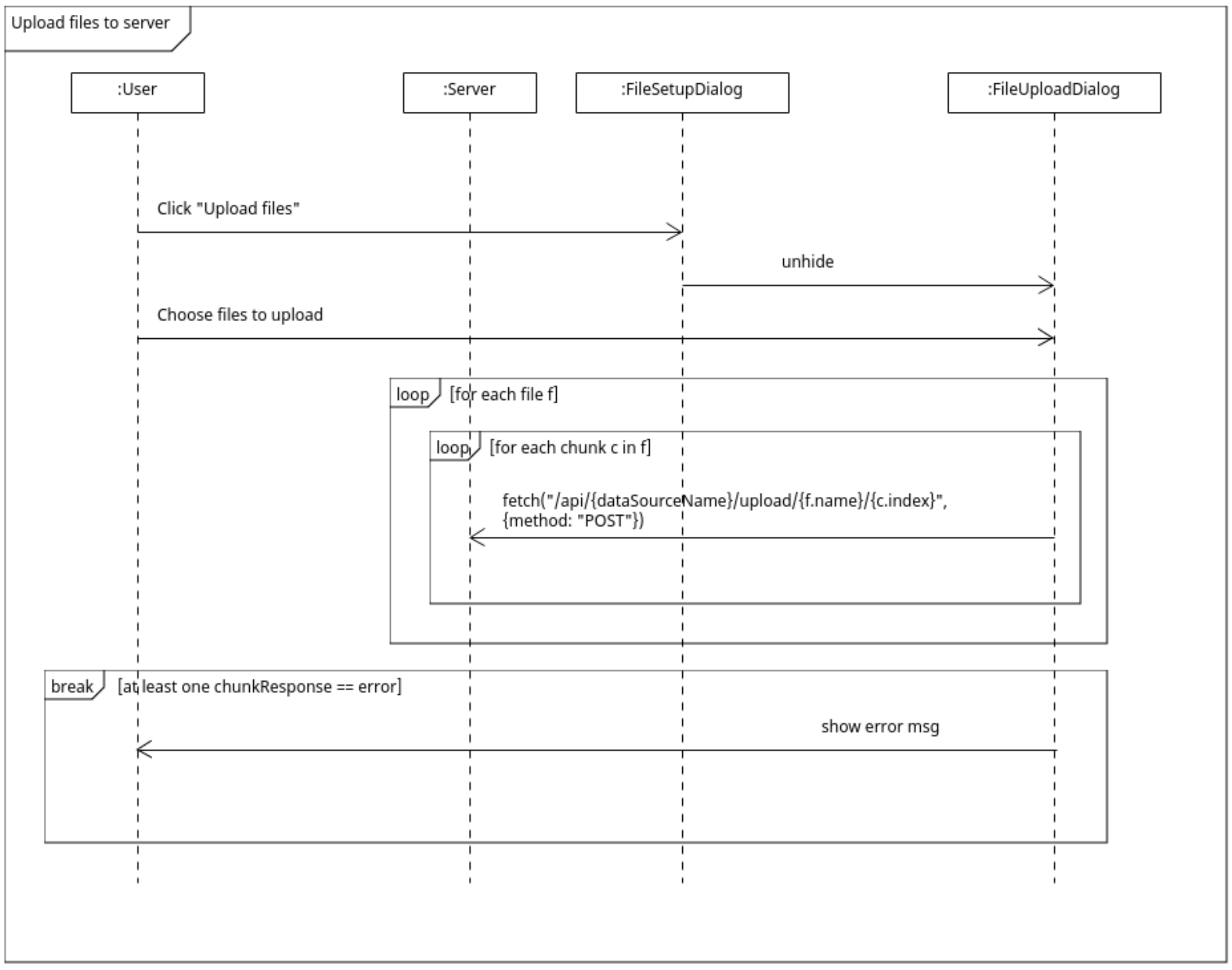

Figure 14: D. Upload a data source

3.3.2 Delete a data source

To delete a data source, the user clicks the delete data source button. In the pop up the user has to confirm the deletion and optionally select whether all the associated files have to be deleted or not. If the user confirms, the system will delete the data source.

Goal	To delete a data source
Preconditions	Data source is loaded and the workspace is opened.
Postconditions	The data source is deleted.
User	All
Priority	Should have

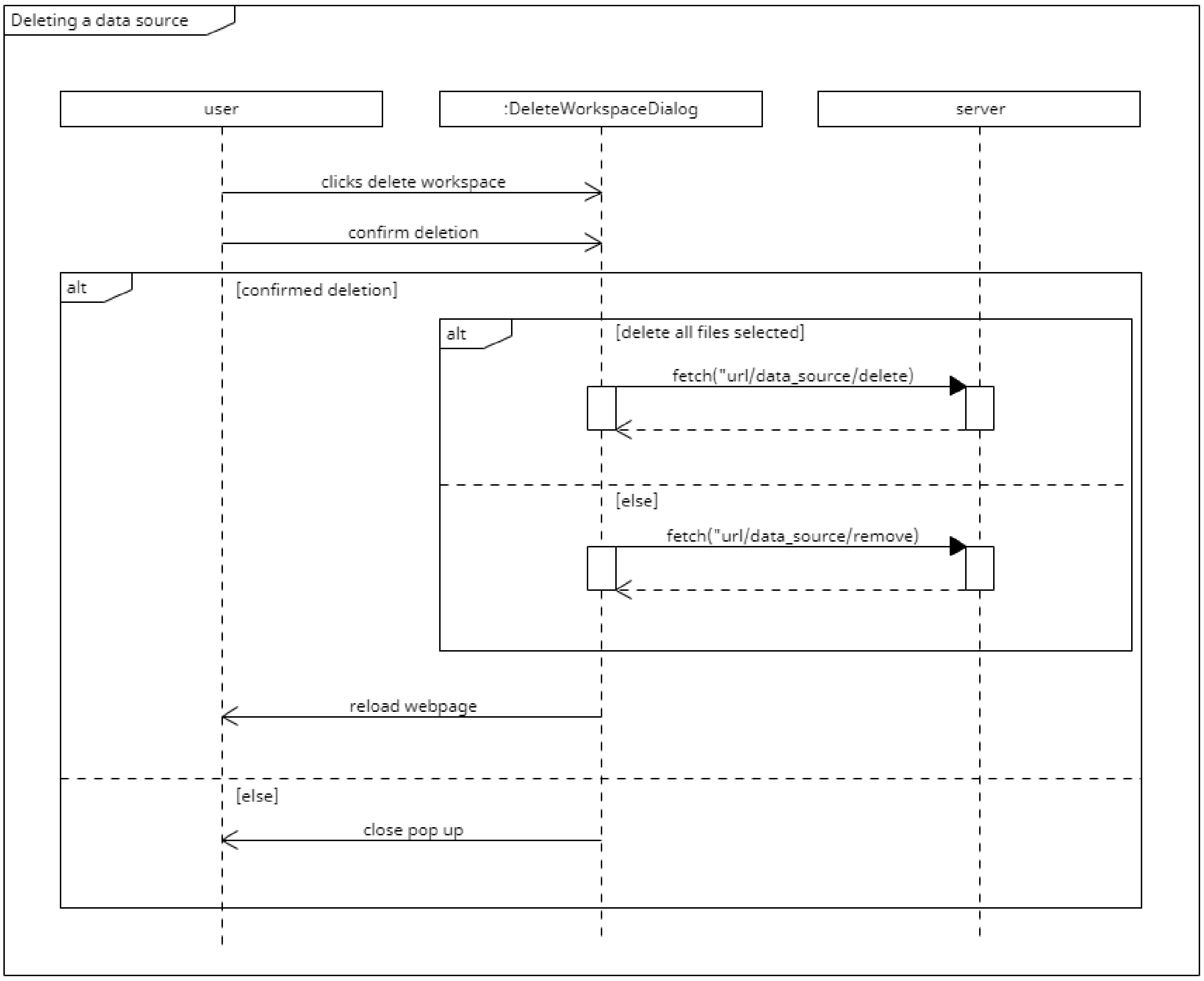

Figure 15: Delete a data source

3.3.3 Add data to existing data source

To add data to an existing data source, the user opens the ”Workspace” window and then clicks on the ”Configure” icon of the base image. A pop-up will appear, where the user clicks ”Upload files” to upload all relevant files. Then, the user chooses the new data source elements by mapping them to the uploaded files. Finally, the user clicks on the ”Save” button”, and the system adds the new data to the data source.

Goal	To upload new data into existing data source
Preconditions	There must be an already existing data source
Postconditions	The new data is added to the existing data source
User	All
Priority	Must have

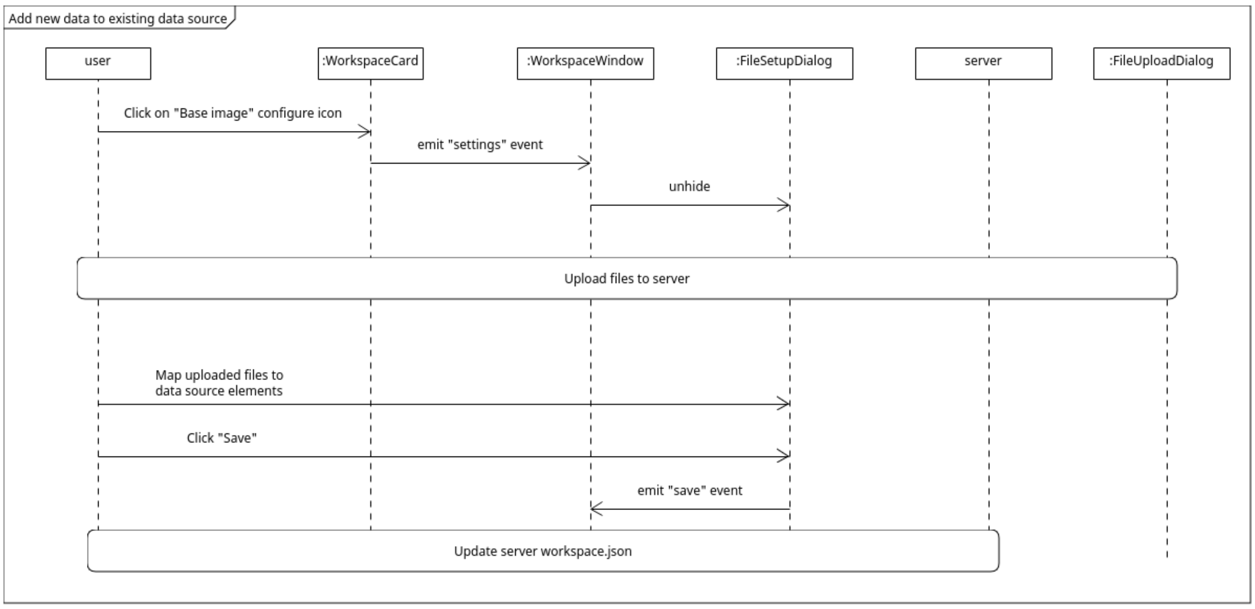

Figure 16: A. Add data to existing data source

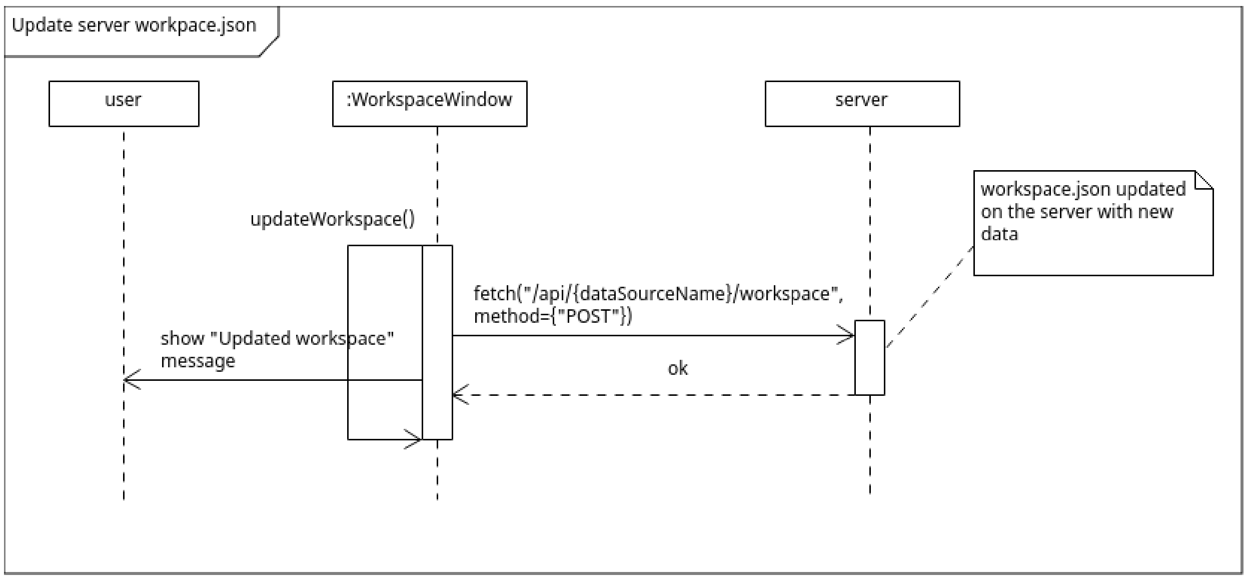
Update server workpace.json
user
:WorkspaceWindow
server
updateWorkspace()
fetch("/api/{dataSourceName}/workspace", method={"POST"})
ok
show "Updated workspace" message
workspace.json updated on the server with new data

Figure 17: B. Add data to existing data source

3.3.4 Delete data from existing data source

To delete data from an existing data source, the user opens the ”Workspace” window and then clicks on the ”Configure” icon of the base image. A pop-up will appear, where the user clicks on the delete icon on each element to be removed. Finally, the user clicks on the ”Save” button”, and the system removes the data from the data source.

Goal	To delete data from existing data source
Preconditions	There must be an already existing data source
Postconditions	The data is deleted from the existing data source
User	All
Priority	Must have

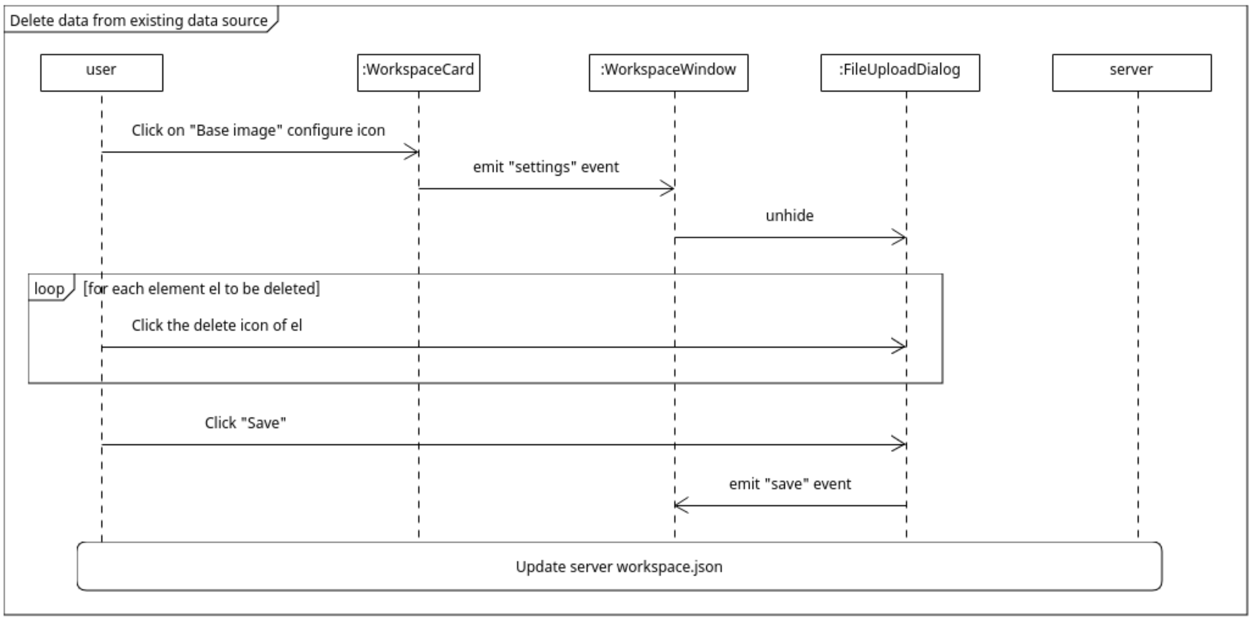
Delete data from existing data source
user
:WorkspaceCard
:WorkspaceWindow
:FileUploadDialog
server
Click on "Base image" configure icon
emit "settings" event
unhide
loop
[for each element el to be deleted]
Click the delete icon of el
Click "Save"
emit "save" event
Update server workspace.json

Figure 18: Delete data from existing data source

3.3.5 Edit data source elements

To edit an existing data source, the user opens the ”Workspace” window and then clicks on the ”Configure” icon of the base image. A pop-up will appear, where the user can change the names of each element and change the file associated with each element. Finally, the user clicks on the ”Save” button”, and the system saves the changes on the data source.

Goal	To edit data source elements
Preconditions	There must be an already existing data source
Postconditions	The changes made by the user are saved
User	All
Priority	Must have

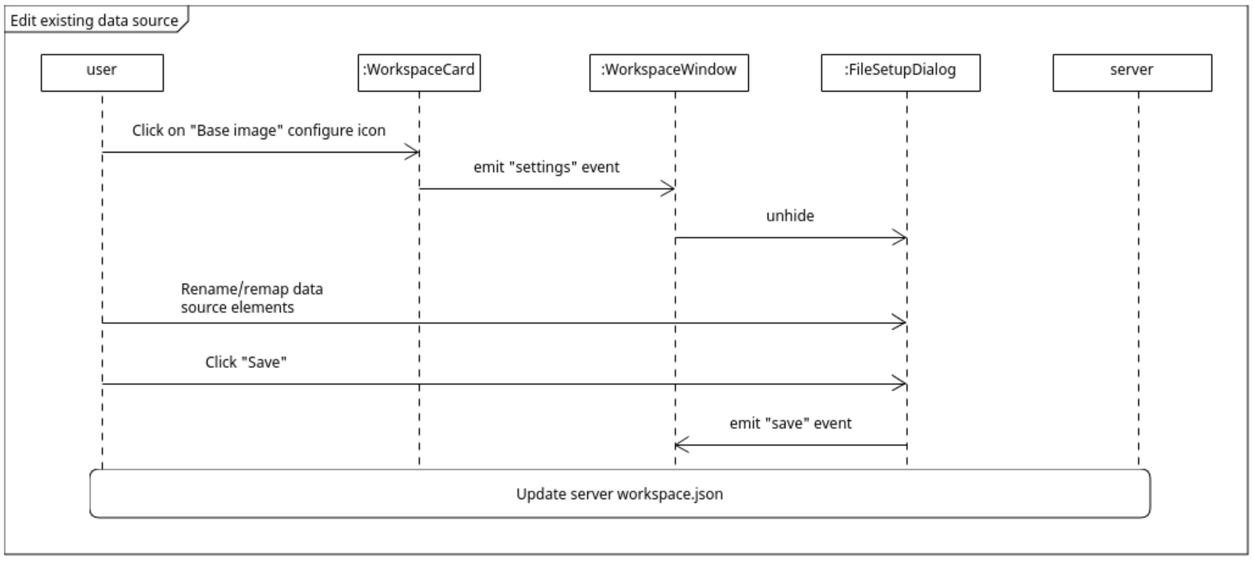
Edit existing data source
user
:WorkspaceCard
:WorkspaceWindow
:FileSetupDialog
server
Click on "Base image" configure icon
emit "settings" event
unhide
Rename/remap data source elements
Click "Save"
emit "save" event
Update server workspace.json

Figure 19: Edit data source elements

3.3.6 Edit the elemental channels of existing data source

To edit the elemental channels of an existing data source, the user opens the ”Workspace” window and then clicks on the ”Configure” icon of ”Generated data”. A pop-up will appear, where the user can toggle which elemental channels are to be included in the workspace. Finally, the user clicks on the ”Save” button”, and the system saves the elemental channel changes for the data source.

Goal	To edit data source elemental channels
Preconditions	There must be an already existing data source
Postconditions	The changes made by the user are saved
User	All
Priority	Must have

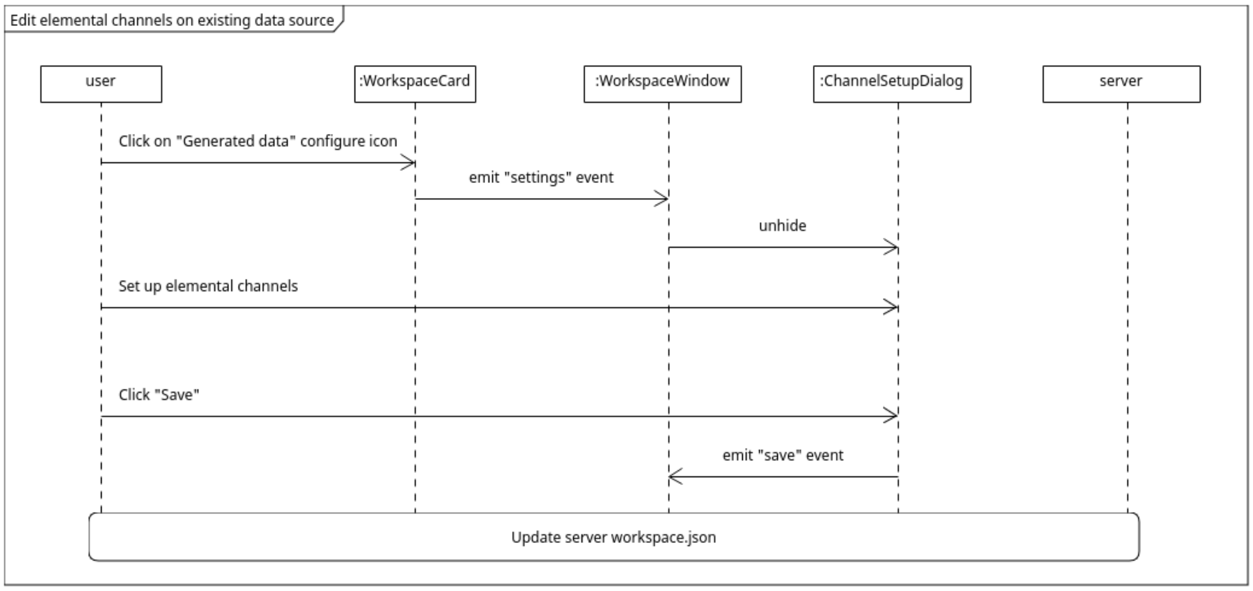
Edit elemental channels on existing data source
user
:WorkspaceCard
:WorkspaceWindow
:ChannelSetupDialog
server
Click on "Generated data" configure icon
emit "settings" event
unhide
Set up elemental channels
Click "Save"
emit "save" event
Update server workspace.json

Figure 20: Edit elemental channels of data source

3.3.7 Load a data source

To load a data source, the user selects ”File” and then clicks on the desired data source name. After a brief period, the system will display the selected data source.

Goal	To load a data source
Preconditions	There must be an already existing data source
Postconditions	The chosen data source is loaded
User	All
Priority	Must have

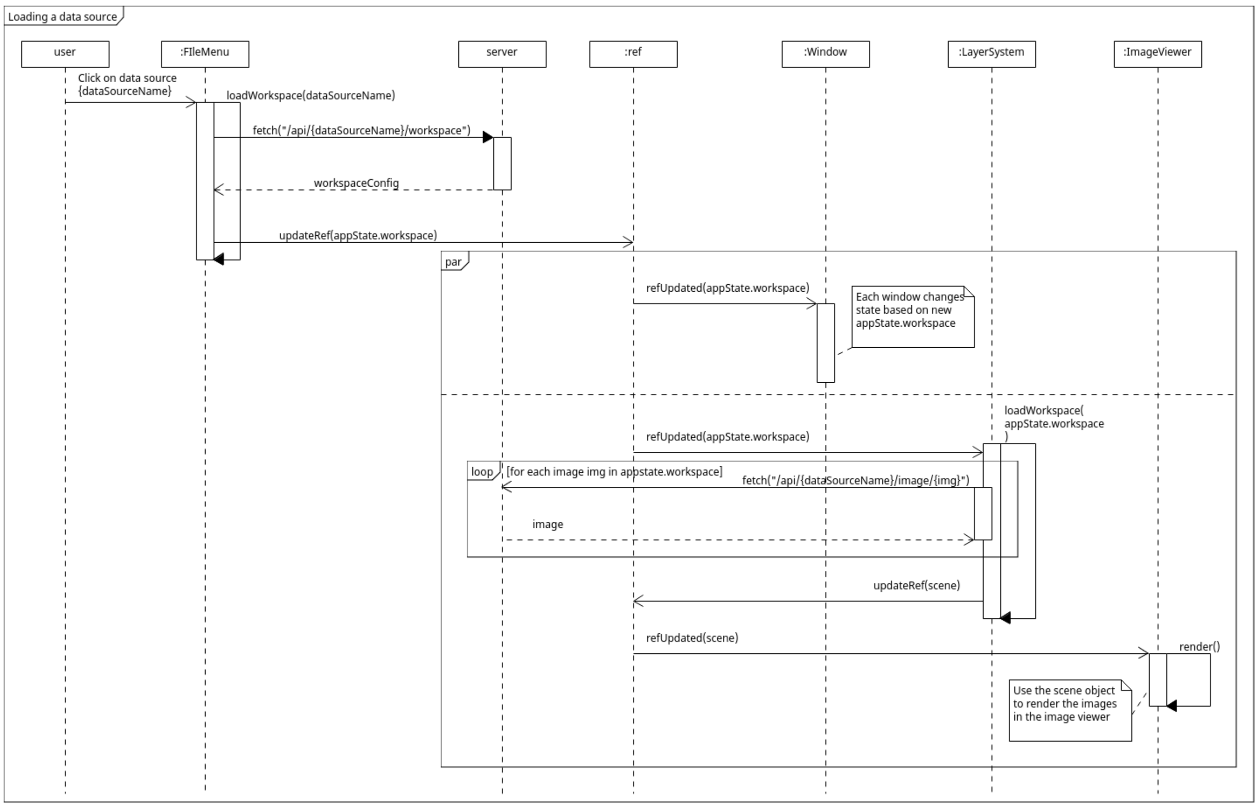
Loading a data source
user
:FIleMenu
server
:ref
:Window
:LayerSystem
:ImageViewer
Click on data source {dataSourceName}
loadWorkspace(dataSourceName)
fetch("/api/{dataSourceName}/workspace")
workspaceConfig
updateRef(appState.workspace)
par
refUpdated(appState.workspace)
Each window changes state based on new appState.workspace
loadWorkspace(appState.workspace)
refUpdated(appState.workspace)
loop
[for each image img in appstate.workspace]
fetch("/api/{dataSourceName}/image/{img}")
image
updateRef(scene)
refUpdated(scene)
render()
Use the scene object to render the images in the image viewer

Figure 21: Load a data source

3.3.8 Enable the second main viewer

To enable the second main viewer, the user only has to enable it through the ”View” pop-up menu in the Header. This is done by checking the checkbox.

Goal	Enable the second main viewer which allows the user to visualize the painting in multiple areas simultaneously.
Preconditions	Only one main viewer is enabled
Postconditions	The second main viewer is displayed to the user
User	All
Priority	Could have

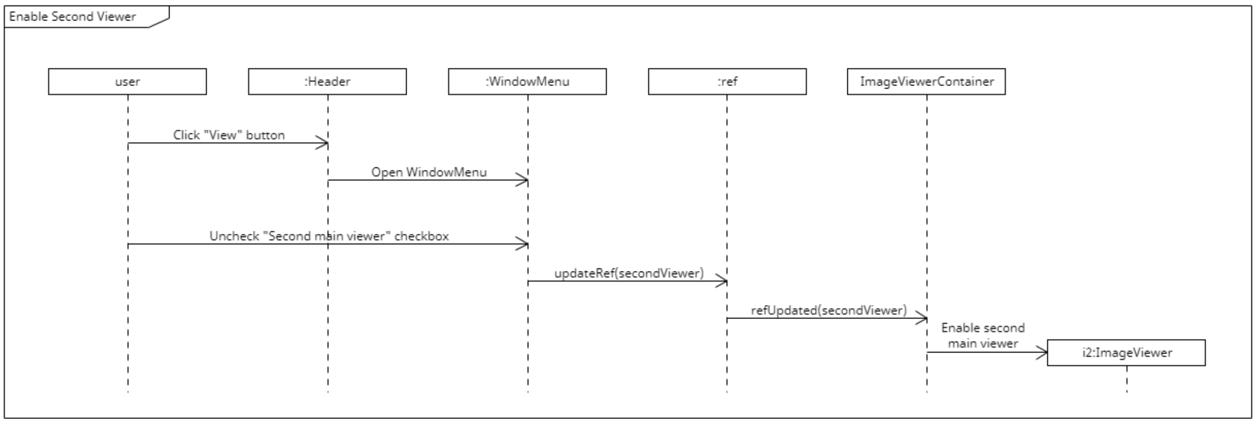

Figure 22: Sequence diagram for enabling the second main viewer

3.3.9 Disable the second main viewer

To disable the second main viewer, the user only has to disable it through the ”View” pop-up menu in the Header. This is done by unchecking the checkbox.

Goal	Disabling the second main viewer to allow for a larger display space for the single main viewer.
Preconditions	The second main viewer is enabled
Postconditions	Only one main viewer is displayed to the user
User	All
Priority	Could have

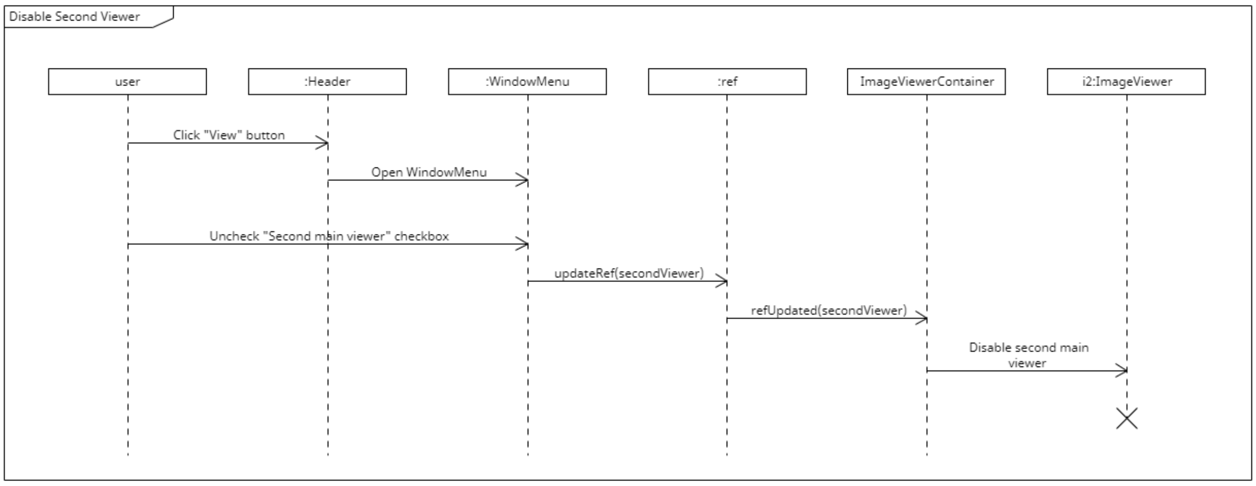

Figure 23: Sequence diagram for disabling the second main viewer

3.3.10 Panning in the main viewer

To pan in the main viewer, the user has to click and hold on the main viewer. The user can then freely drag the image around to pan. Upon realising the left mouse button, moving the mouse will no longer pan the image viewer.

Goal	Allowing the user to freely pan in the image viewer allowing them to explore all areas of the images.
Preconditions	The panning tool is selected in the toolbar
Postconditions	The image in the main viewer is panned accordingly with the user's input
User	All
Priority	Must have

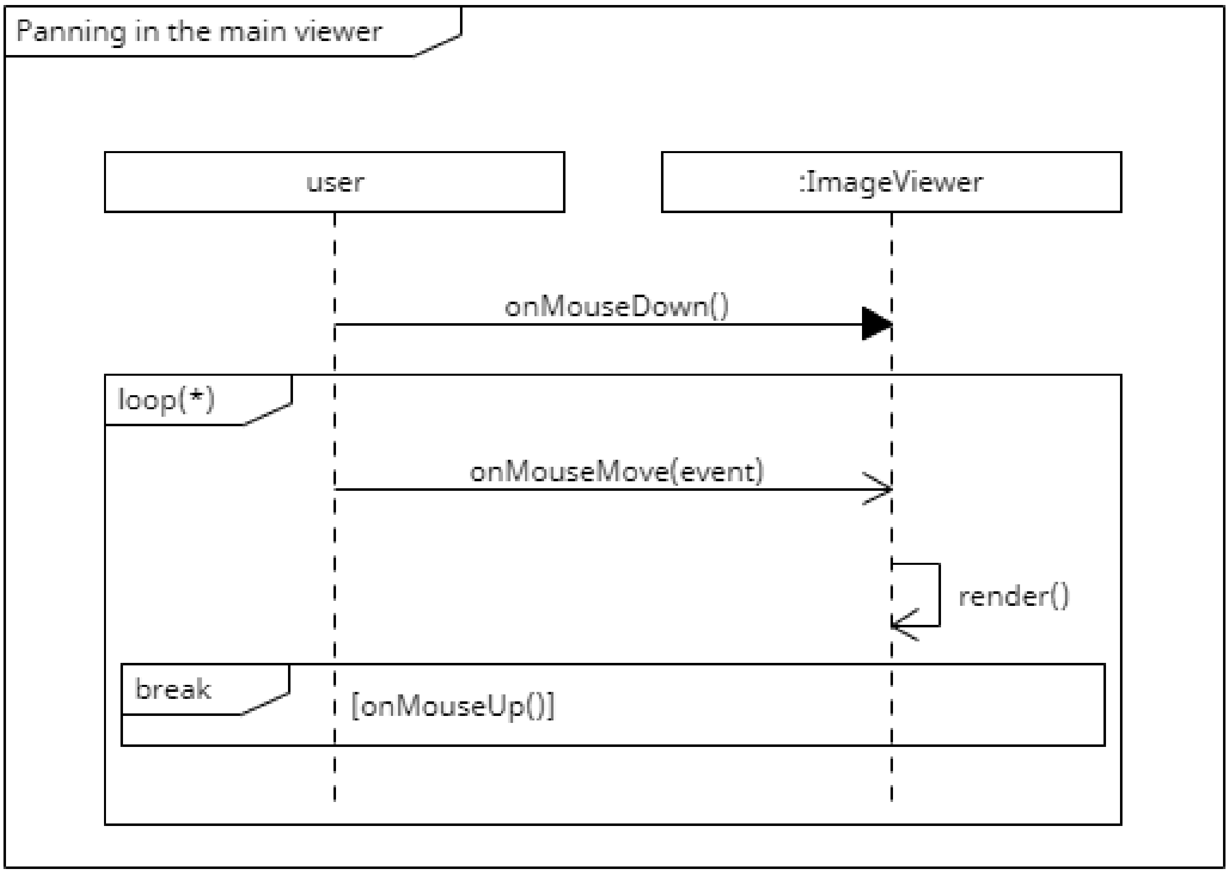

Figure 24: Sequence diagram for panning in the main viewer

3.3.11 Zooming in the main viewer

To zoom in or out on the main viewer, the user has to make a scroll motion using their mouse (pinching won't work for touchpads).

Goal	To allow users to freely zoom in on the image viewer allowing them to zoom in on details.
Preconditions	A data source has been loaded in.
Postconditions	The image in the main viewer is zoomed in/out accordingly with the user's input. If the zoom reaches the maximum zoom level, the zoom is clamped.
User	All
Priority	Must have

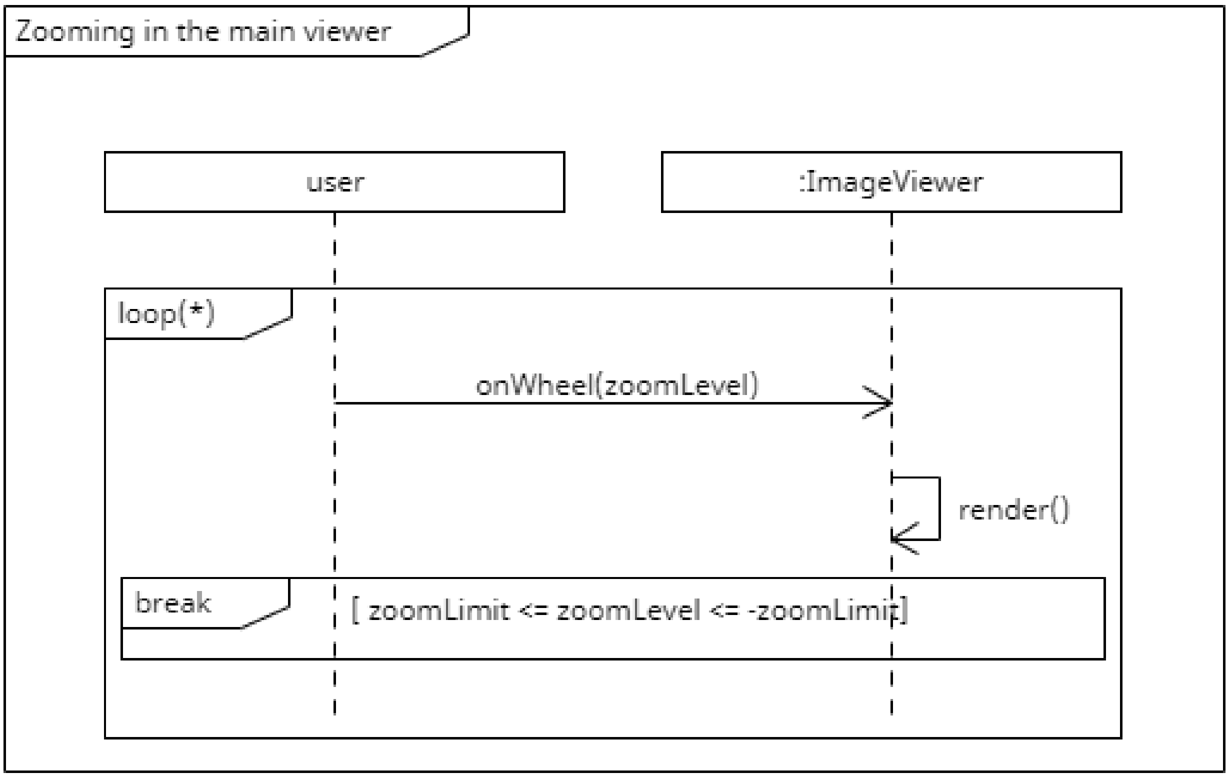
Zooming in the main viewer
user
:ImageViewer
loop(*)
onWheel(zoomLevel)
render()
break
[zoomLimit <= zoomLevel <= -zoomLimit]

Figure 25: Sequence diagram for zooming in the main viewer

3.3.12 Export main viewer

When the user requests to export the main viewer, the system will first create a renderer. The system will then retrieve the necessary information for exporting the image in the main viewer, after which it renders the image and saves it.

Goal	Saving the image present in the main viewer.
Preconditions	The main viewer is enabled.
Postconditions	The image in the main viewer is saved on the user's device.
User	All
Priority	Should have

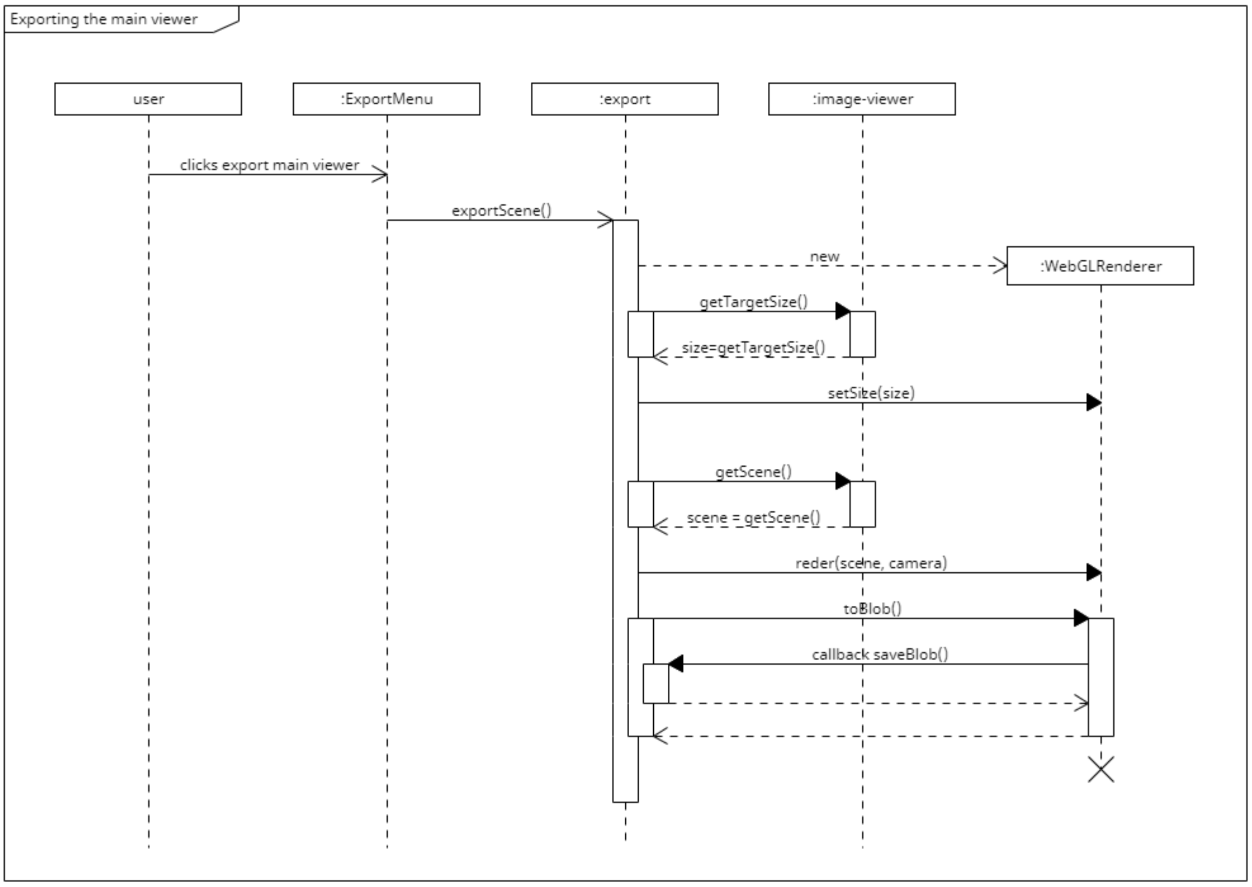

Figure 26: Exporting the main viewer.

3.3.13 Export a visualization

When the user requests to export a visualization, the system will convert the html element to an image and save it to the user's device.

Goal	To save a visualization to the user's device.
Preconditions	The main viewer is enabled and the visualization to be exported is present.
Postconditions	The visualization the user requested to export is saved on the user's device.
User	All
Priority	Should have

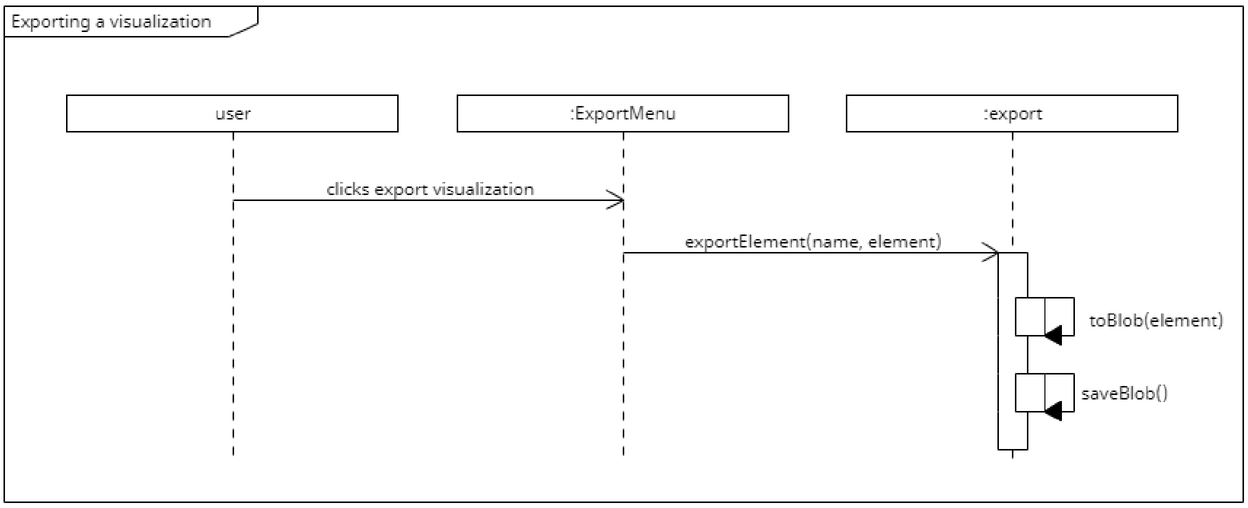
Exporting a visualization
user
:ExportMenu
:export
clicks export visualization
exportElement(name, element)
toBlob(element)
saveBlob()

Figure 27: Exporting a visualization.

3.3.14 Make lasso selection

To create a lasso selection, the user has to click on an object that supports selection making. When the user has clicked, the point is added to the selection. In case the point is close to the first point in the selection, the system will complete the selection by notifying the ref.

Goal	Creating a lasso selection and updating all visualizations using that selection. This sequence diagram refers to both the main viewer and the dimensionality reduction window.
Preconditions	The main viewer is enabled and the lasso tool has been selected.
Postconditions	The selection is updated with the added point.
User	All
Priority	Must have

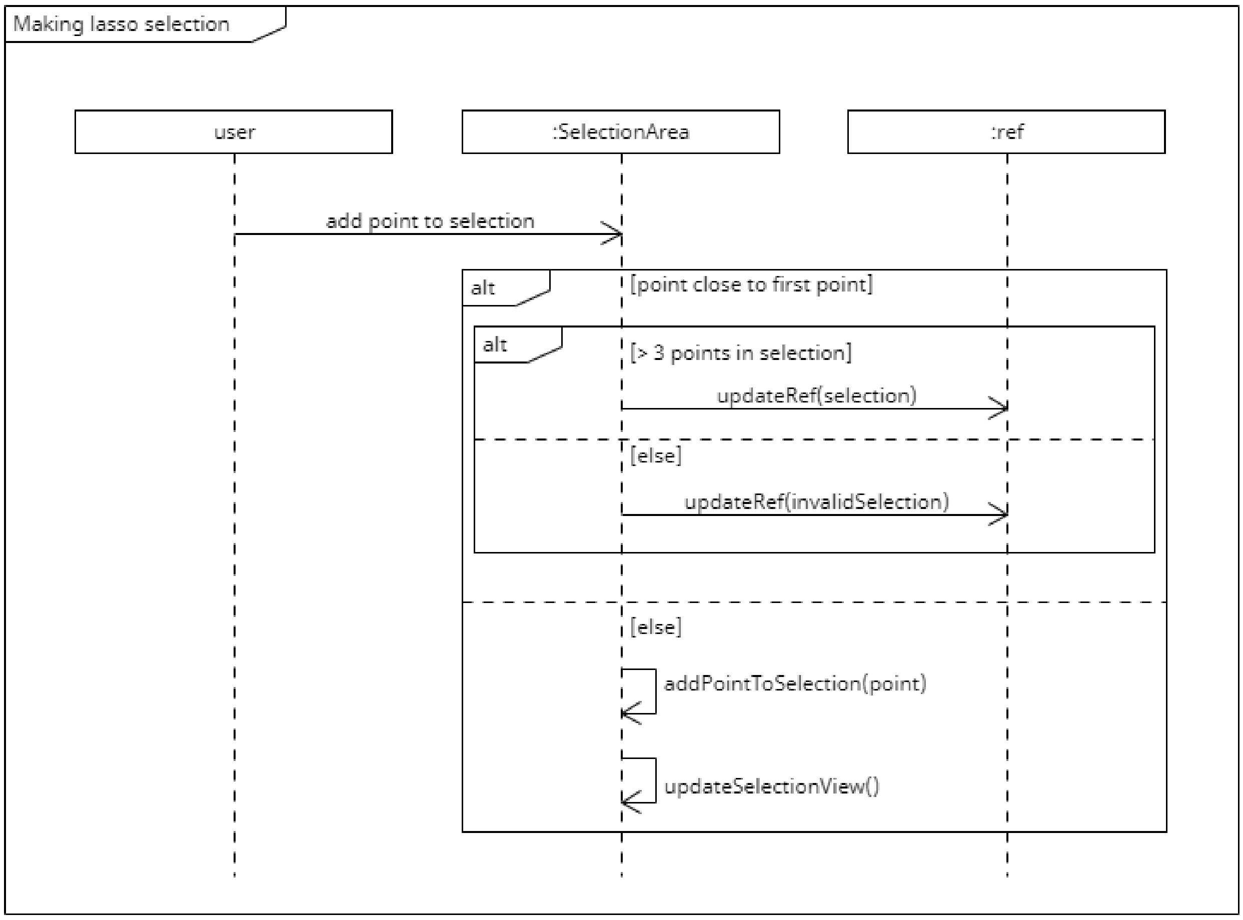

Figure 28: Sequence diagram for making a lasso selection

3.3.15 Make rectangle selection

To create a rectangle selection, the user has to click and hold on an object that supports selection making. When the user has clicked and not released, the point is added to the selection. Next, when the user moves their mouse the selection visualization on the object will update. Finally, when the user releases their mouse, the system will complete the selection by notifying the ref.

Goal	Creating a rectangle selection and updating all visualizations using that selection. This sequence diagram refers to both the main viewer and the dimensionality reduction window.
Preconditions	The main viewer is enabled and the rectangle tool has been selected.
Postconditions	The selection is updated with the added point.
User	All
Priority	Must have

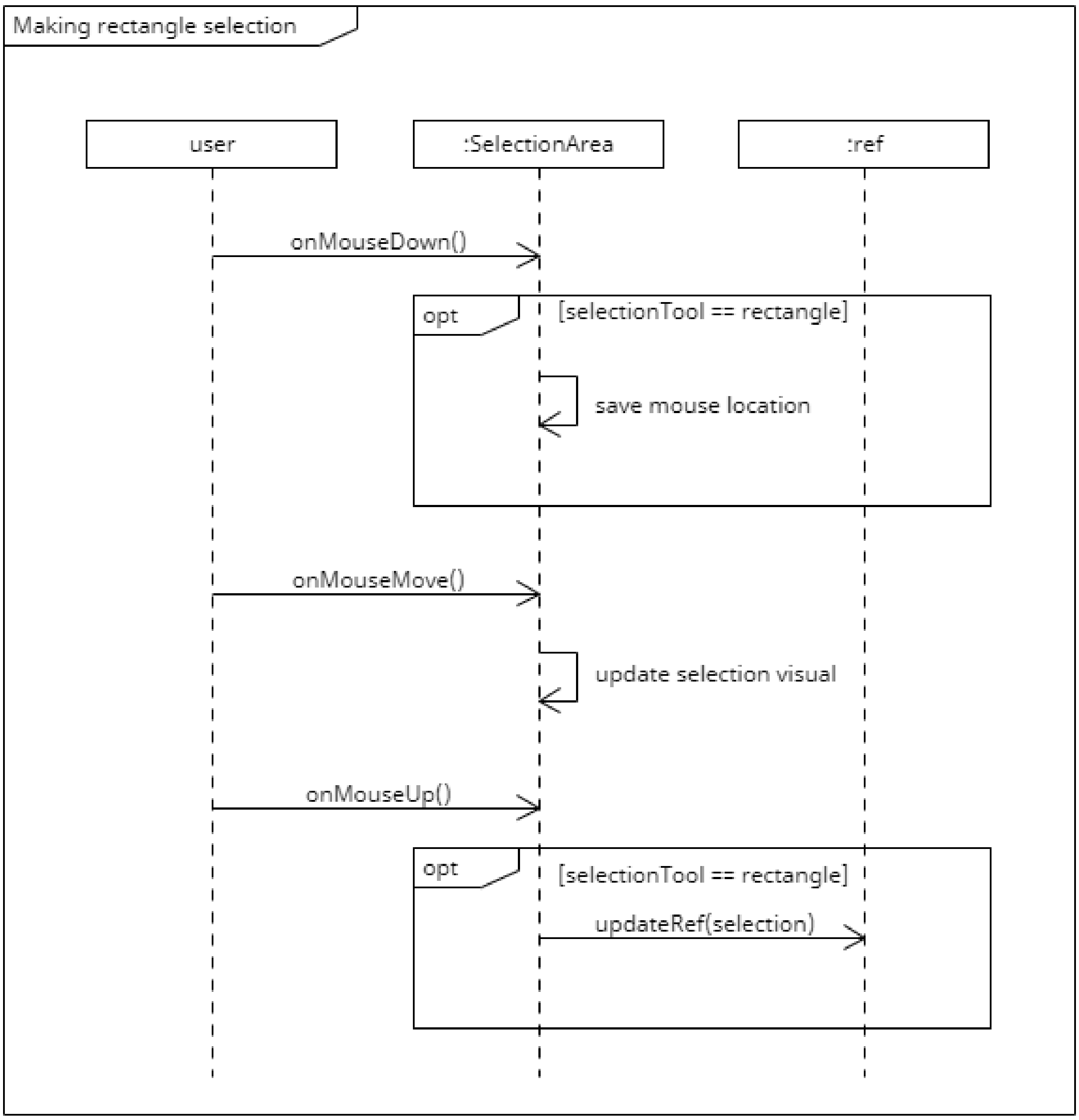

Figure 29: Sequence diagram for making a rectangle selection

3.3.16 Enabling a layer

To enable a layer, the user has to request to enable a layer. The system then enables the layer and updates the main viewer.

Goal	To enable a layer.
Preconditions	The layer that the user wants is disabled and the user is in the layer window.
Postconditions	The requested layer to enable is enabled.
User	All
Priority	Must have

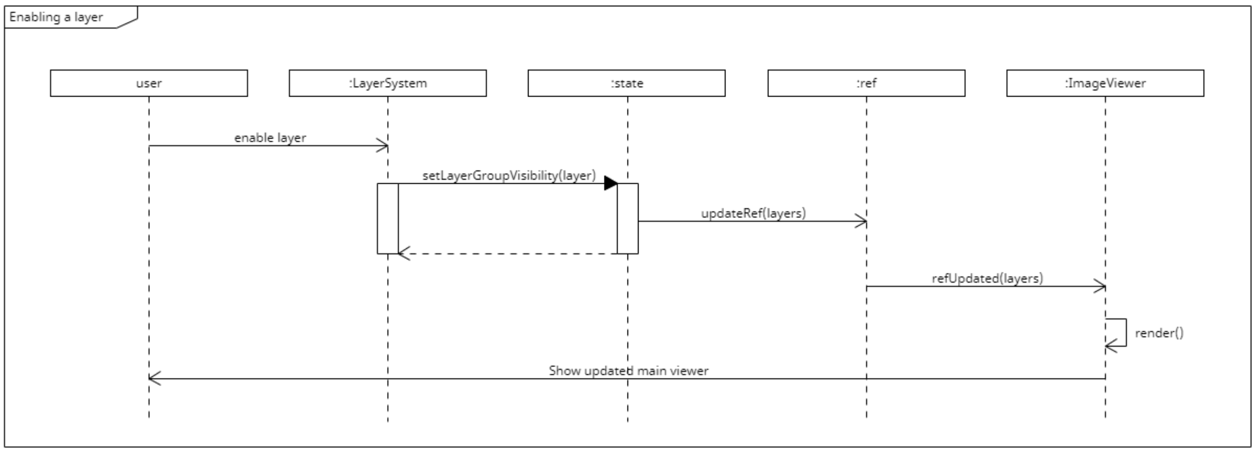

Figure 30: Sequence diagram for enabling a layer

3.3.17 Changing the layer hierarchy

To move a layer and update the hierarchy, the user has to first move the layer. The system then updates the layer position, updates the hierarchy of the layers and finally updates the main viewer.

Goal	To move a layer.
Preconditions	The user is in the layer window.
Postconditions	The requested layer to move has been moved and the layer hierarchy is updated.
User	All
Priority	Must have

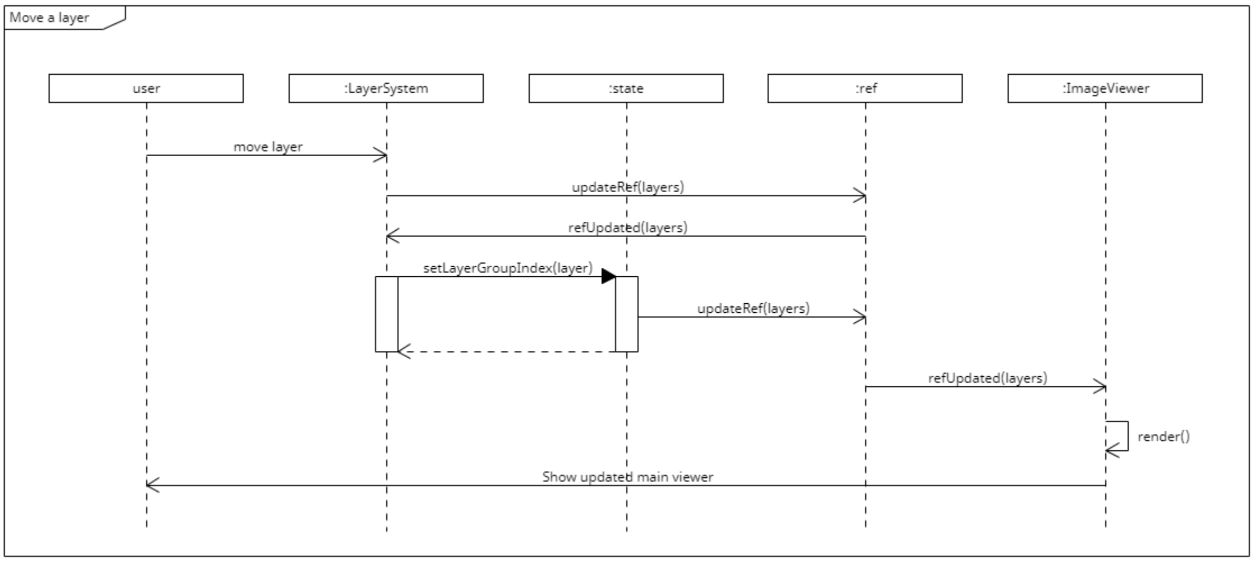

Figure 31: Sequence diagram for moving a layer

3.3.18 Change the opacity of a layer

To change the opacity of a layer, the layer has to be enabled. Then the user can simply change the opacity value of the specific layer by using the opacity slider.

Goal	Changing the opacity of a specific layer.
Preconditions	The layer whose opacity should be changed is enabled
Postconditions	The layer's opacity is updated and displayed in the image viewer
User	All
Priority	Must have

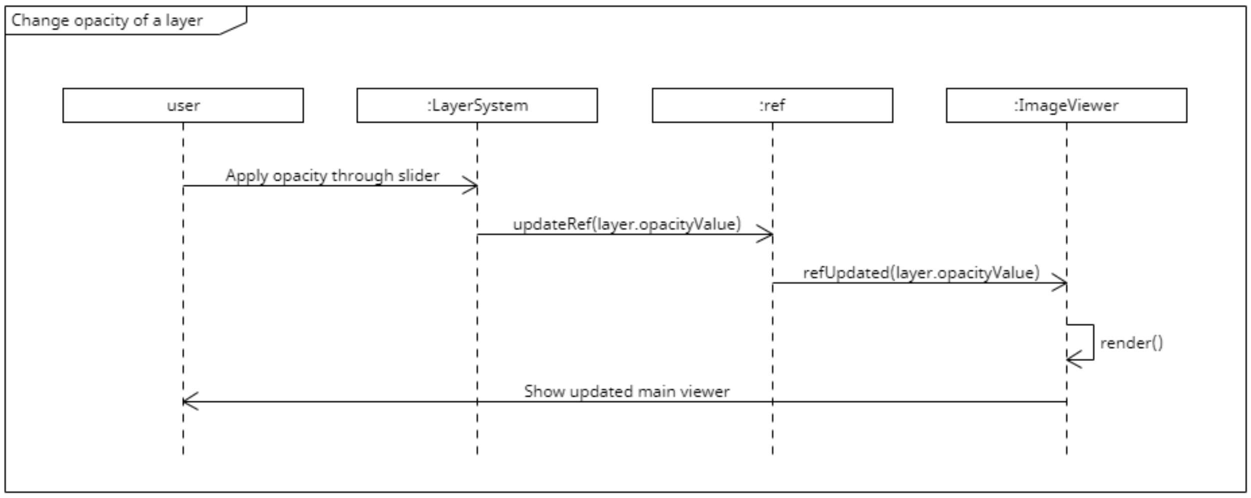
Change opacity of a layer
user
:LayerSystem
:ref
:ImageViewer
Apply opacity through slider
updateRef(layer.opacityValue)
refUpdated(layer.opacityValue)
render()
Show updated main viewer

Figure 32: Sequence diagram for changing the opacity

3.3.19 Change the filter value of a layer

To change the filter value of a layer, the user first has to open the filter slider menu. From this menu, the user is able to change the value of each filter individually through the usage of a slider.

Goal	To change the filter applied to the layer.
Preconditions	The layer whose filter value should be changed is enabled.
Postconditions	The layer's filter is updated and displayed in the image viewer.
User	All
Priority	Must/should have

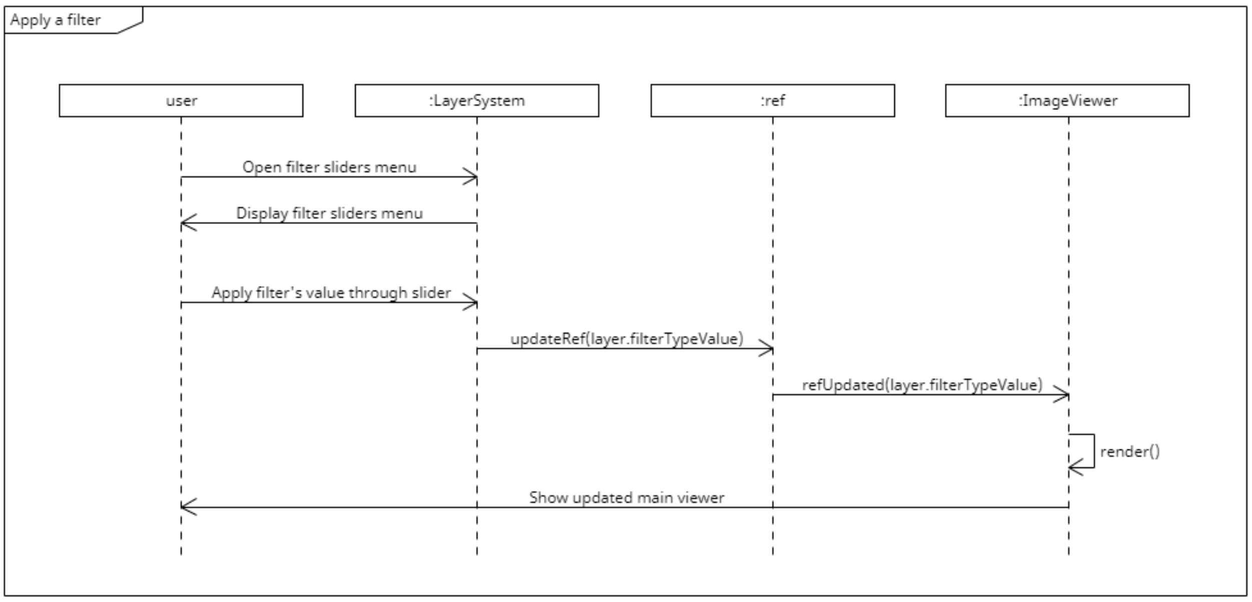
Apply a filter
user
:LayerSystem
:ref
:ImageViewer
Open filter sliders menu
Display filter sliders menu
Apply filter's value through slider
updateRef(layer.filterTypeValue)
refUpdated(layer.filterTypeValue)
render()
Show updated main viewer

Figure 33: Sequence diagram for applying a filter value

3.3.20 Resetting the filter value of a layer

To reset the filter value of a layer, the user first has to open the filter slider menu. From this menu, the user is able to reset the value of each filter individually by double-clicking on its sliders.

Goal	To reset the filter applied to the layer.
Preconditions	The layer whose filter value should be reset is enabled.
Postconditions	The layer's filter is reset and displayed in the image viewer.
User	All
Priority	Could have

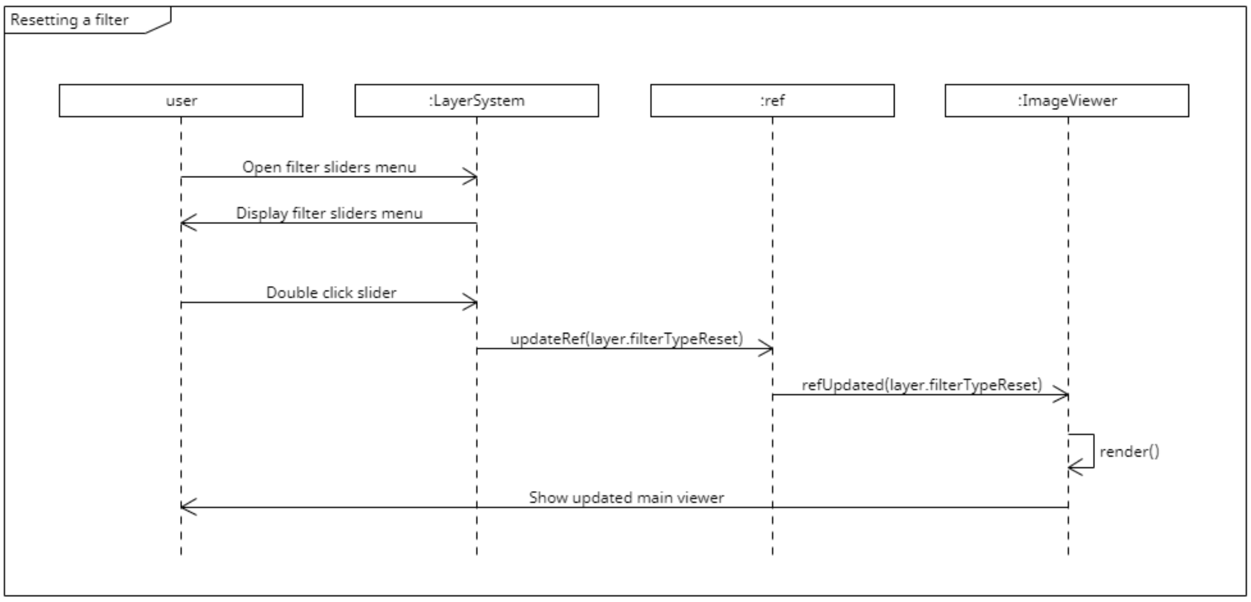

Figure 34: Sequence diagram for resetting a filter value

3.3.21 Enabling the lens and choosing a layer

When the user wants to use the lens, they first have to select the layer(s) they want to view inside the lens. To do this, the user has to select the option in the layer system to view the layer inside the lens. The system will then update the visibility of the selected layer. Next, the user selects the lens tool from the toolbar. The system will then update the main viewer to show the selected layer inside the lens.

Goal	To use the lens tool.
Preconditions	Lens tool is disabled, layer window is enabled with at least one layer.
Postconditions	Lens tool is enabled and the selected layer is being shown inside the lens.
User	All
Priority	Must Have

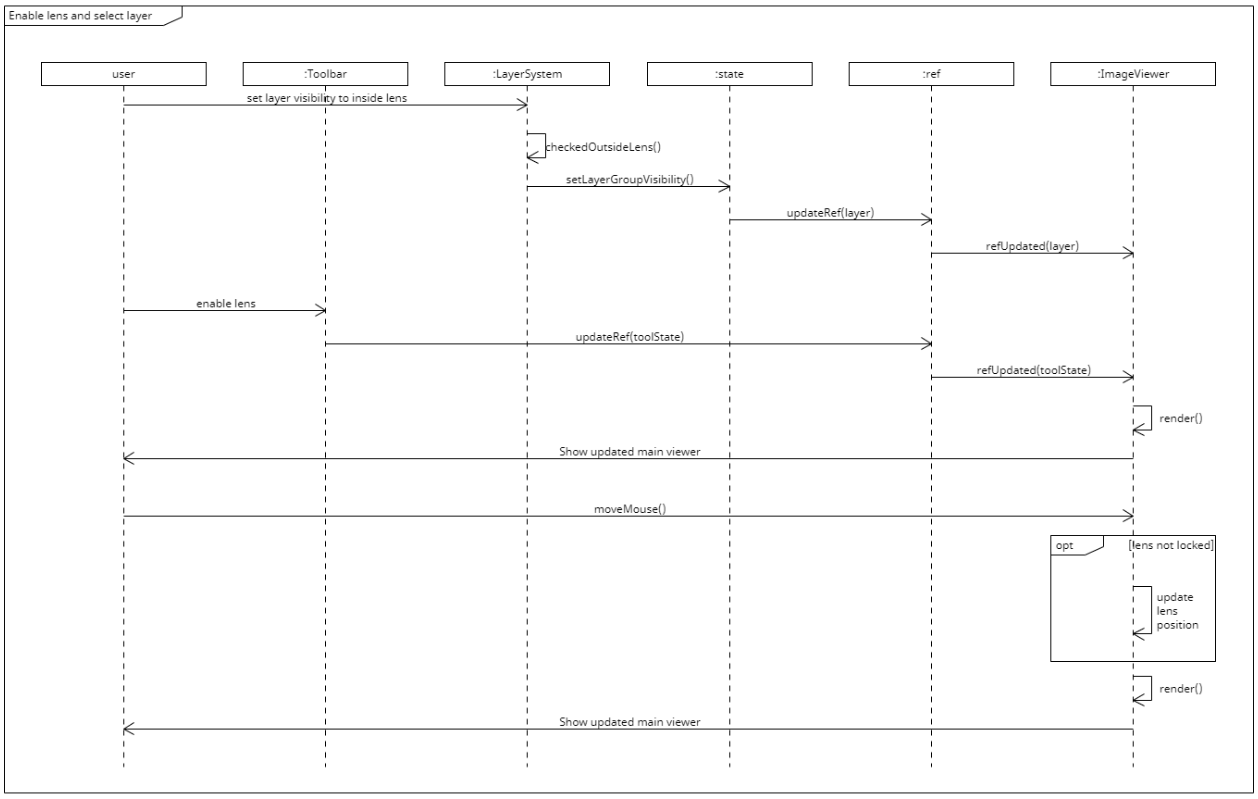

Figure 35: Sequence diagram for enabling the lens

3.3.22 Locking the lens

When the user wants to lock the lens in place, they have to press the right mouse button on the main viewer. Then the system will set the lens to locked. When moving the mouse, the system will then not change the position of the lens.

Goal	To lock the lens.
Preconditions	Lens tool is enabled and layer window is enabled.
Postconditions	Lens is locked in place when moving the mouse.
User	All
Priority	Should Have

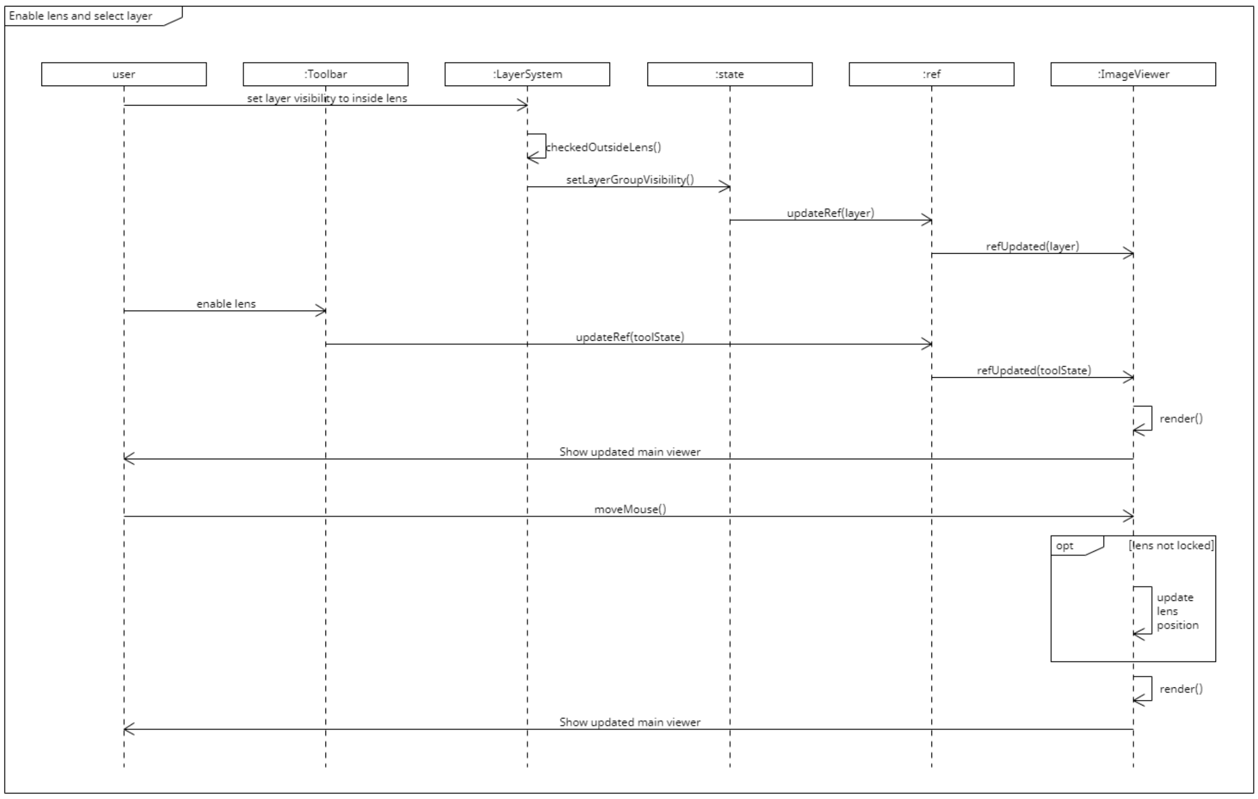

Figure 36: Sequence diagram for enabling the lens

3.3.23 Generate spectra chart of the global average

To generate the spectra chart for the global average, the user opens the elemental charts window. Then the system loads the global averages in the background. When the user selects the global average, the system will create the chart and display it to the user.

Goal	Generate and display the spectra chart of the global averages for the entire painting.
Preconditions	Raw data cube available.
Postconditions	Global averages are generated and a chart with the global averages is being shown to the user.
User	All
Priority	Must Have

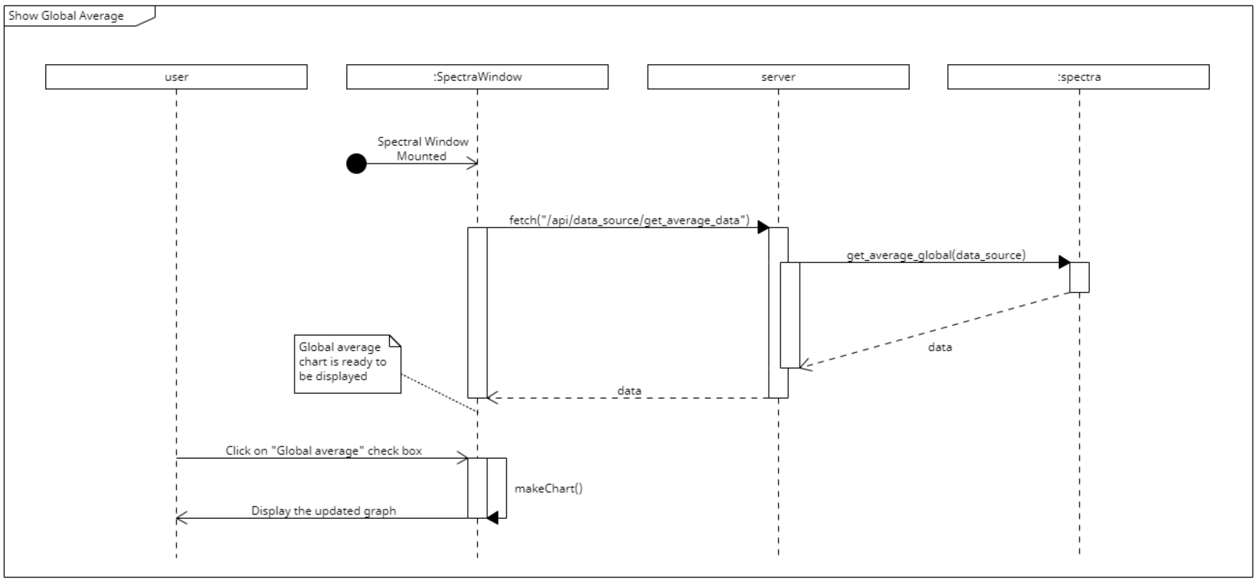
Show Global Average
user
:SpectraWindow
server
:spectra
Spectral Window Mounted
fetch("/api/data_source/get_average_data")
get_average_global(data_source)
data
Global average chart is ready to be displayed
data
Click on "Global average" check box
makeChart()
Display the updated graph

Figure 37: Sequence diagram for generating spectra chart of the global average

3.3.24 Generate theoretical spectra chart of a selected element

To generate the theoretical spectra chart of an element, the user first selects the show theoretical elemental spectrum. The system then adds the theoretical element line to the chart. Next, the user selects an element and a excitation value. If both are selected, the system will fetch the theoretical data of the selected element and update the chart.

Goal	Generate and display the theoretical spectra chart of the selected element.
Preconditions	Raw data cube available.
Postconditions	Line chart with the theoretical spectra chart of the selected element is displayed.
User	All
Priority	Must Have

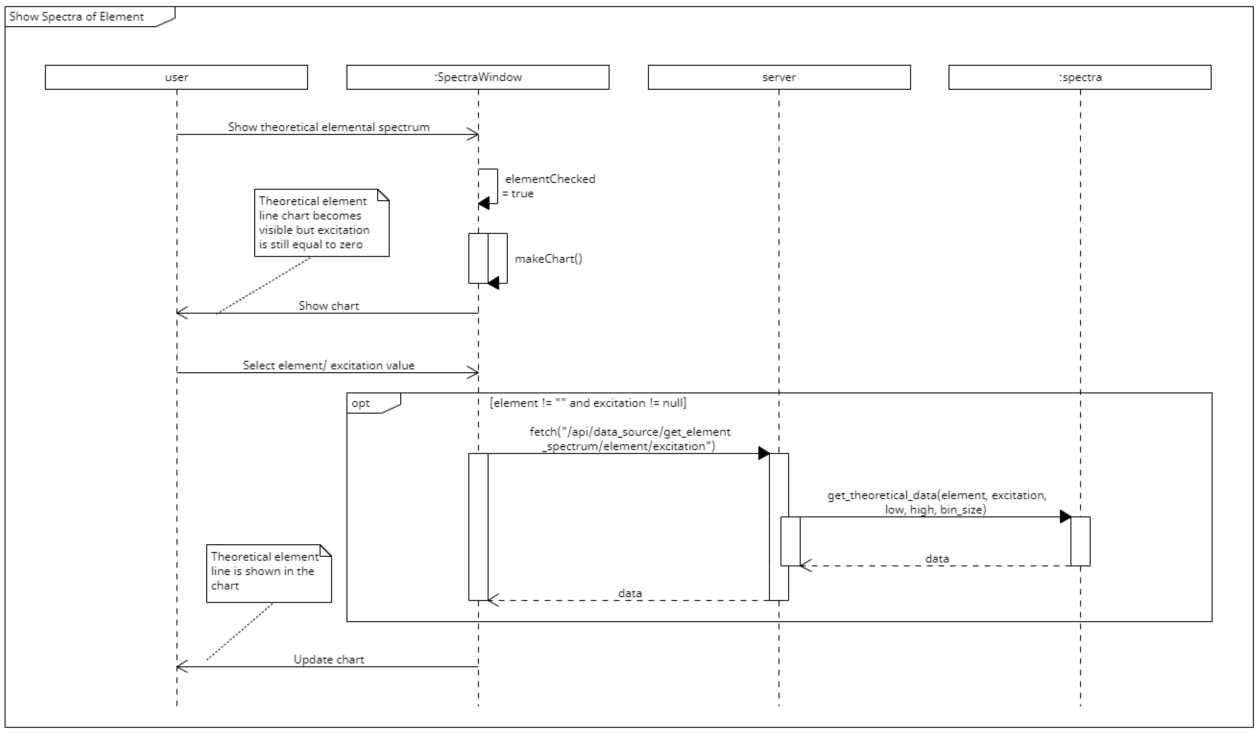
Show Spectra of Element
user
:SpectraWindow
server
:spectra
Show theoretical elemental spectrum
elementChecked = true
Theoretical element line chart becomes visible but excitation is still equal to zero
makeChart()
Show chart
Select element/ excitation value
opt
[element != "" and excitation != null]
fetch("/api/data_source/get_element_spectrum/element/excitation")
get_theoretical_data(element, excitation, low, high, bin_size)
data
Theoretical element line is shown in the chart
data
Update chart

Figure 38: Sequence diagram for generating the theoretical spectra chart of a selected element

3.3.25 Generate spectra within selection

To generate the spectra chart within a selection, the user first selects an area in the main viewer. Then the system fetches the averages within the area selected. Afterwards, the system updates the chart with the spectra of the selection.

Goal	Generate spectra chart within a selection made in the main viewer.
Preconditions	Raw data cube available
Postconditions	Line chart with the spectra within the selected area is displayed.
User	All
Priority	Must Have

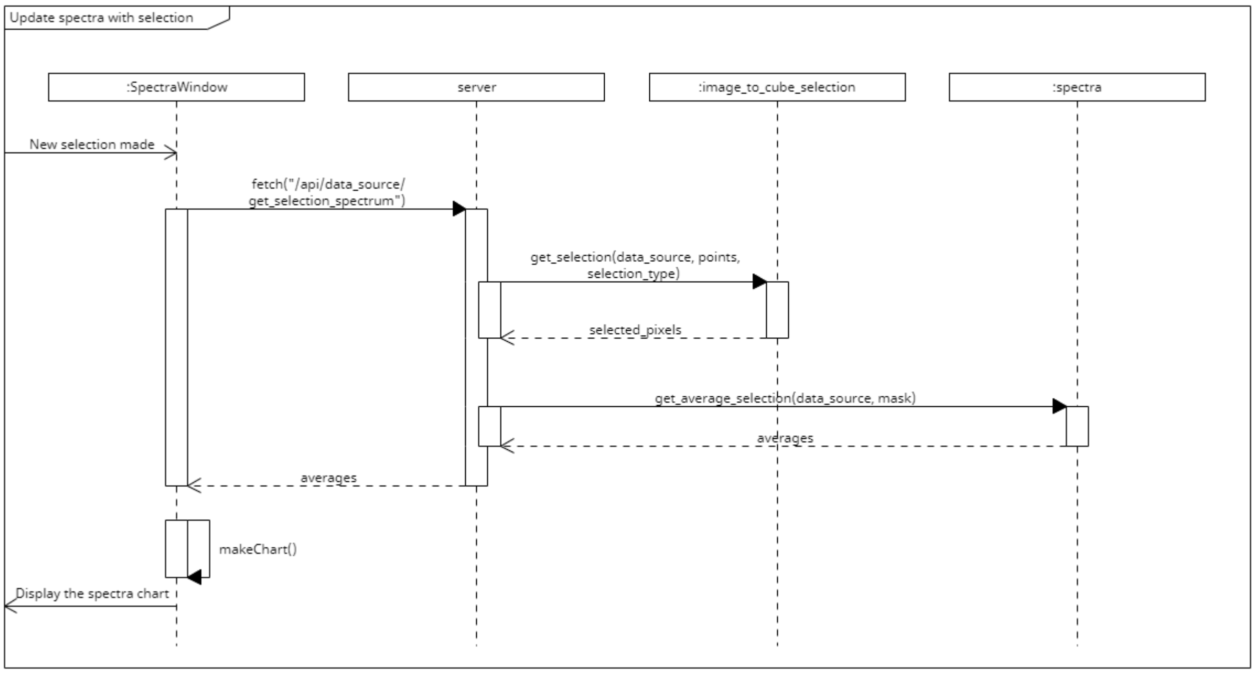
Update spectra with selection
:SpectraWindow
server
:image_to_cube_selection
:spectra
New selection made
fetch("/api/data_source/get_selection_spectrum")
get_selection(data_source, points, selection_type)
selected_pixels
get_average_selection(data_source, mask)
averages
averages
makeChart()
Display the spectra chart

Figure 39: Sequence diagram for generating the spectra chart for the selected area

3.3.26 Generate bar chart with the average abundance of every element

To generate a bar chart with the average abundance of every element, the user needs to open the elemental charts window. When the window is opened, the system will fetch the average abundance of every element. Afterwards, the system updates/creates the bar chart of the average abundance.

Goal	Generate bar chart with the average abundance of every element.
Preconditions	Elemental data cube is available
Postconditions	Bar chart with the average abundance of every element is displayed.
User	All
Priority	Must Have

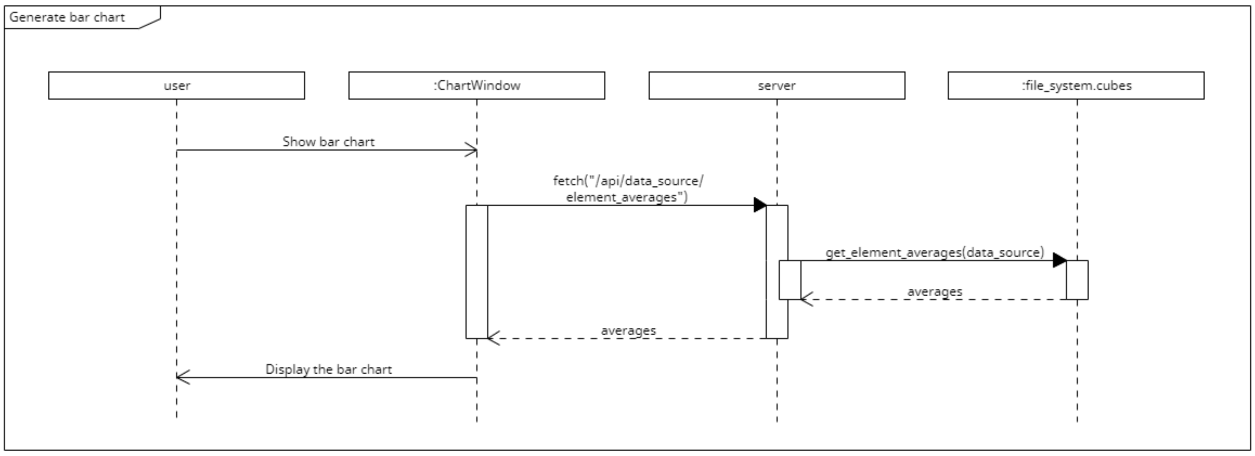

Figure 40: Sequence diagram for generating bar chart with the average abundance of every element.

3.3.27 Generate line chart with the area selected by the user

To generate a line chart of the average abundance in the selected area, the AppState reports that a new selection has been made to the ChartWindow. Then, the ChartWindow fetches the average abundance in the selection. Lastly, the system updates the bar chart to display the line of the average abundance in the selection.

Goal	Generate a line chart of the average abundance in the selected area.
Preconditions	Elemental data cube is available
Postconditions	Line chart with the average abundance in the selected area in the main viewer by the user is displayed.
User	All
Priority	Must Have

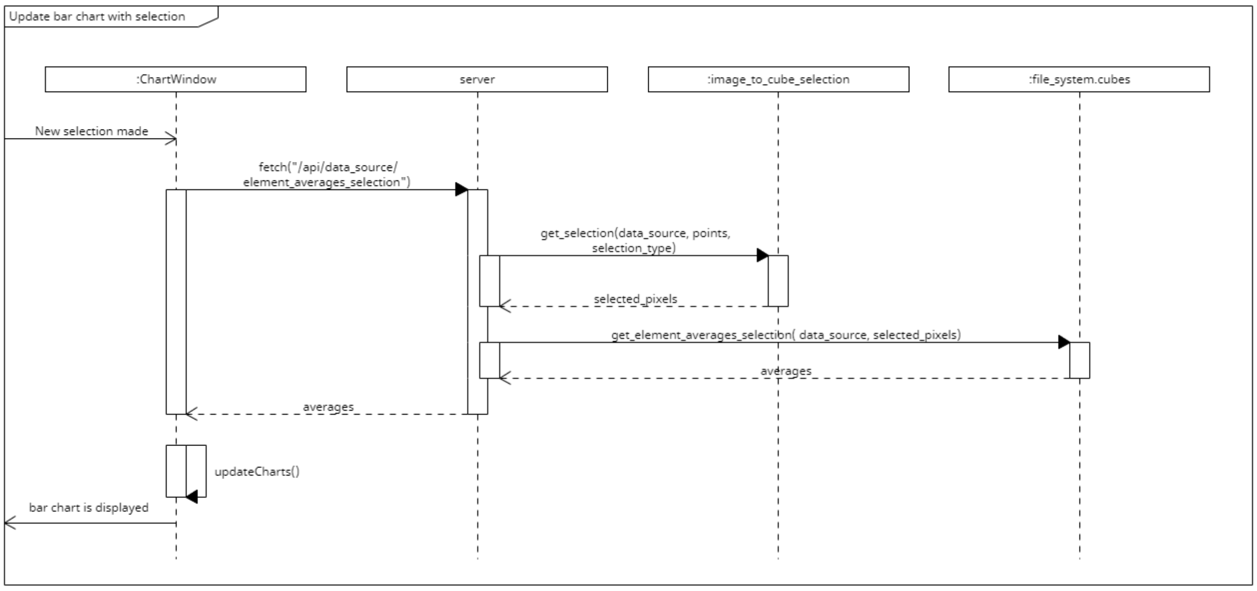
Update bar chart with selection
:ChartWindow
server
:image_to_cube_selection
:file_system.cubes
New selection made
fetch("/api/data_source/element_averages_selection")
get_selection(data_source, points, selection_type)
selected_pixels
get_element_averages_selection(data_source, selected_pixels)
averages
averages
updateCharts()
bar chart is displayed

Figure 41: Sequence diagram for generating line chart with the area selected by the user.

3.3.28 Generate embedding using dimensionality reduction

To generate an embedding, the user selects an element and an threshold. In addition, the user selects an overlay type. After the user clicks on the generate embedding button, the front-end will send a request to the server asking to generate the embedding. If the generation was successful it will load the embedding plot.

Goal	Generate the embedding with the given element and threshold using the dimensionality reduction method
Preconditions	Elemental data cube available
Postconditions	All indices, reduced indices, embedding data, image to map to embedding are stored on the server. Message is sent to user to show the status of the generation.
User	All
Priority	Must Have

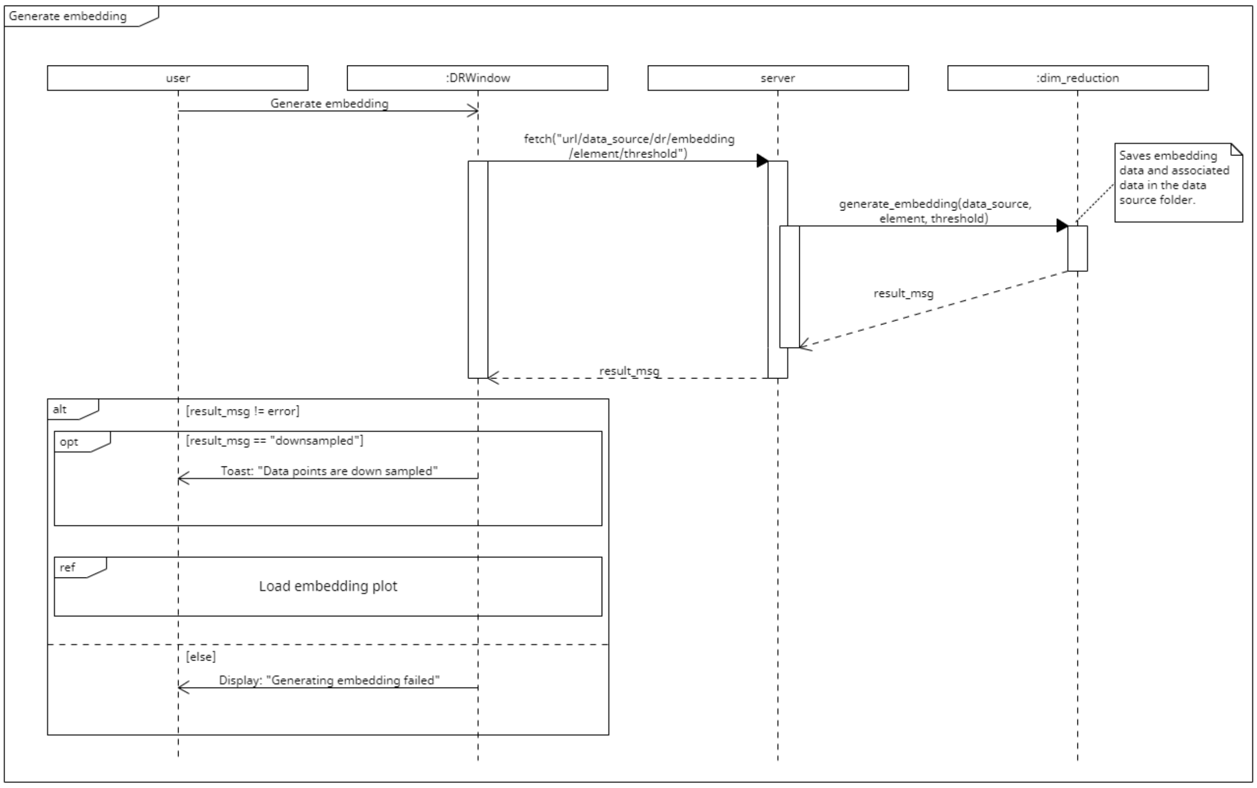
Generate embedding
user
:DRWindow
server
:dim_reduction
Generate embedding
fetch("url/data_source/dr/embedding /element/threshold")
generate_embedding(data_source, element, threshold)
Saves embedding data and associated data in the data source folder.
result_msg
result_msg
alt
[result_msg != error]
opt
[result_msg == "downsampled"]
Toast: "Data points are down sampled"
ref
Load embedding plot
[else]
Display: "Generating embedding failed"

Figure 42: Sequence diagram for generating an embedding

3.3.29 Showing dimensionality reduction embedding with overlay

When the user has generated an embedding for the data source once, they can change the overlay used for the plot. By selecting an overlay type from the dropdown menu, the front-end will fetch the image with the requested overlay type. After some time, the plot is shown. In case of errors the error message is shown.

Goal	To generate the embedding plot
Preconditions	embedding has been generated once and this data is available on the server.
Postconditions	embedding plot is stored on the server and shown to the user. In case of an error, the error is displayed.
User	All
Priority	Must Have

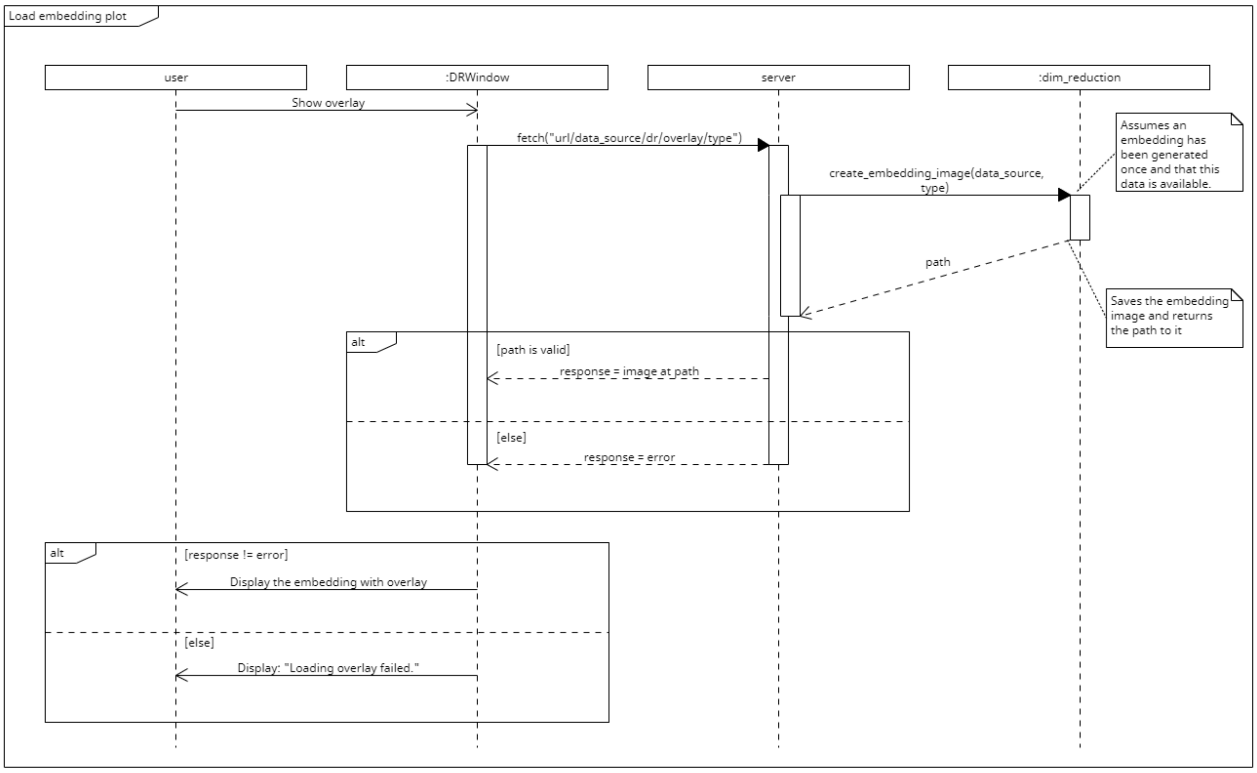
Load embedding plot
user
:DRWindow
server
:dim_reduction
Show overlay
fetch("url/data_source/dr/overlay/type")
create_embedding_image(data_source, type)
Assumes an embedding has been generated once and that this data is available.
path
Saves the embedding image and returns the path to it
alt
[path is valid]
response = image at path
[else]
response = error
alt
[response != error]
Display the embedding with overlay
[else]
Display: "Loading overlay failed."

Figure 43: Sequence diagram for loading and embedding plot

3.3.30 Highlighting the selection in the dimensionality reduction window

First the user has to create a selection. When the selection is made, the system will check whether the pixels in the main viewer corresponding to the pixels in the selection have to be highlighted. In case it does not, the dimensionality reduction layer is disposed. Otherwise, the bitmasks are updated and the dimensionality reduction layer is updated. Finally, the system will render the main viewer and show the result to the user.

Goal	To see the correspondence between the embedding generated by dimensionality reduction and the original image.
Preconditions	embedding has been generated and is being shown, a selection tool in the dimensionality reduction window is enabled, the dimensionality reduction selection layer is enabled.
Postconditions	The pixels in the main viewer corresponding to the pixels inside the selected area are highlighted.
User	All
Priority	Must Have

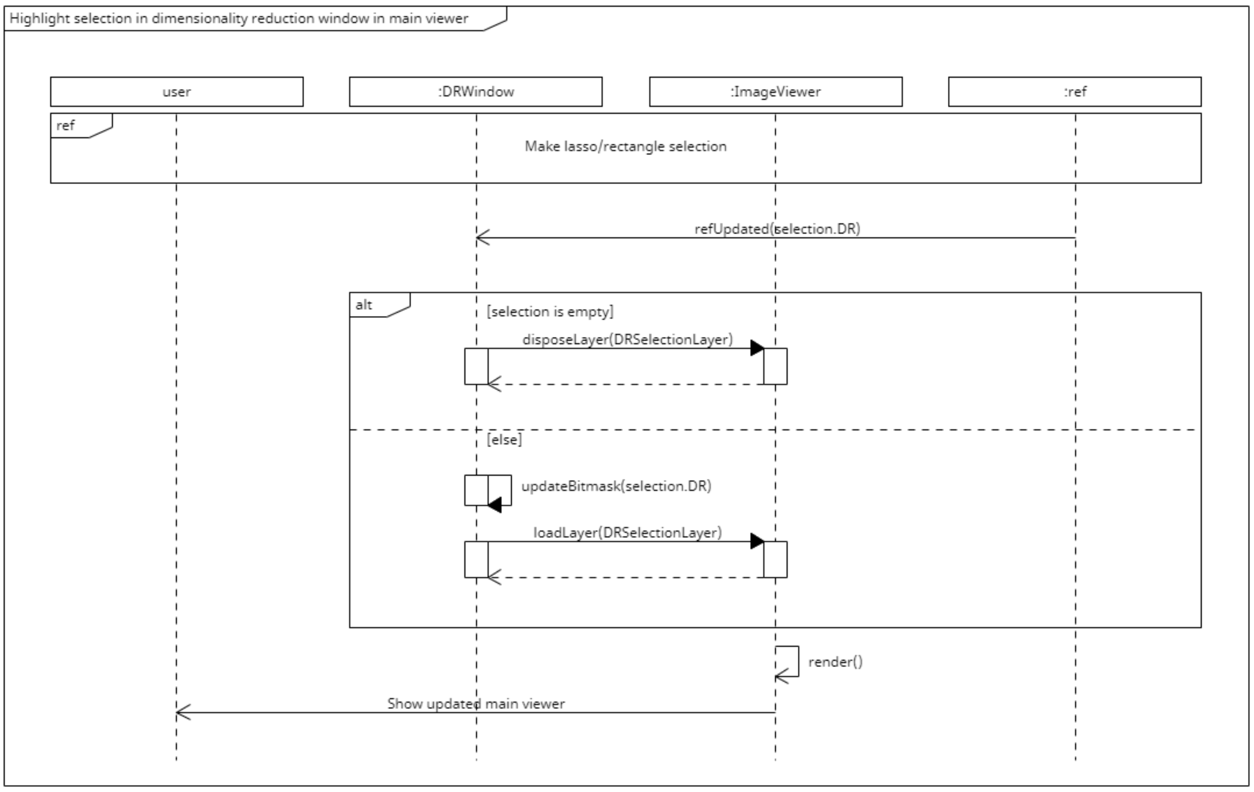
Highlight selection in dimensionality reduction window in main viewer
user
:DRWindow
:ImageViewer
:ref
ref
Make lasso/rectangle selection
refUpdated(selection.DR)
alt
[selection is empty]
disposeLayer(DRSelectionLayer)
[else]
updateBitmask(selection.DR)
loadLayer(DRSelectionLayer)
render()
Show updated main viewer

Figure 44: Sequence diagram for loading and embedding plot

3.3.31 Generating color segmentation with user-given parameters

Computes the color segmentation (CS) colors and bitmask for the user selected element/whole image, caches the bitmask/colors, and displays the colors to the user. If already cached, the colors are simply returned

Goal	To compute the color clusters and bitmasks for the given element/whole image, cache them and send them back to the front-end, or send back directly if already cached
Preconditions	RGB image and elemental data cube available
Postconditions	Color clusters and corresponding bitmask for given parameters are computed and cached, or simply returned if already cached.
User	All
Priority	Must Have

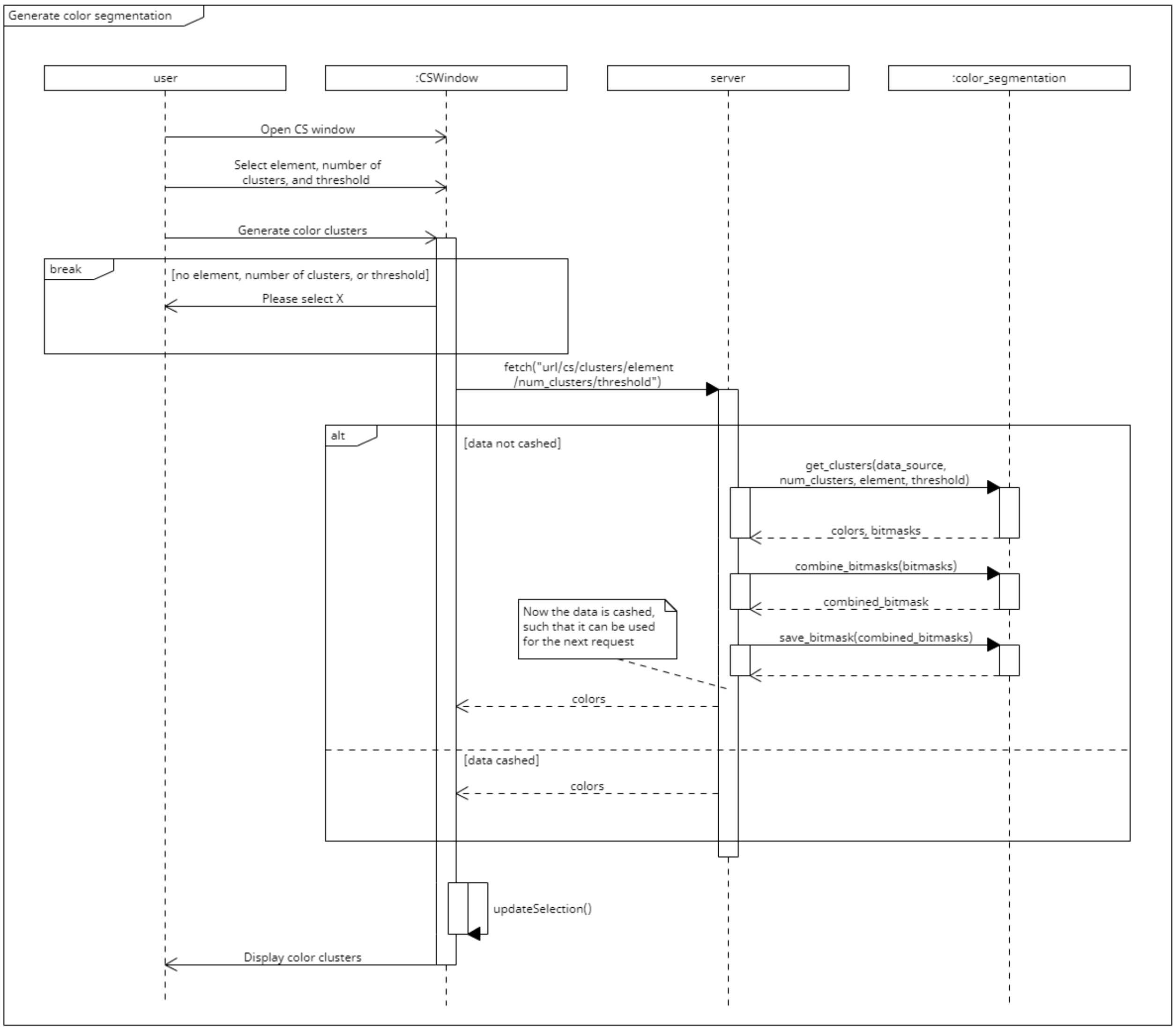
Generate color segmentation
user
:CSWindow
server
:color_segmentation
Open CS window
Select element, number of clusters, and threshold
Generate color clusters
break
[no element, number of clusters, or threshold]
Please select X
fetch("url/cs/clusters/element/num_clusters/threshold")
alt
[data not cashed]
get_clusters(data_source, num_clusters, element, threshold)
colors, bitmasks
combine_bitmasks(bitmasks)
combined_bitmask
save_bitmask(combined_bitmasks)
Now the data is cashed, such that it can be used for the next request
colors
[data cashed]
colors
updateSelection()
Display color clusters

Figure 45: Sequence diagram for loading the CS colors

3.3.32 Enabling an elemental channel

To enable an elemental map, the user has requests to enable the elemental map in the elemental channel window. The system then enabled the elemental channel and updates the main viewer.

Goal	To view an elemental map.
Preconditions	Elemental data cube available, elemental channel window enabled, and elemental map disabled. The elemental layer is enabled in the layer system.
Postconditions	The elemental channel is highlighted in the main viewer.
User	All
Priority	Must Have

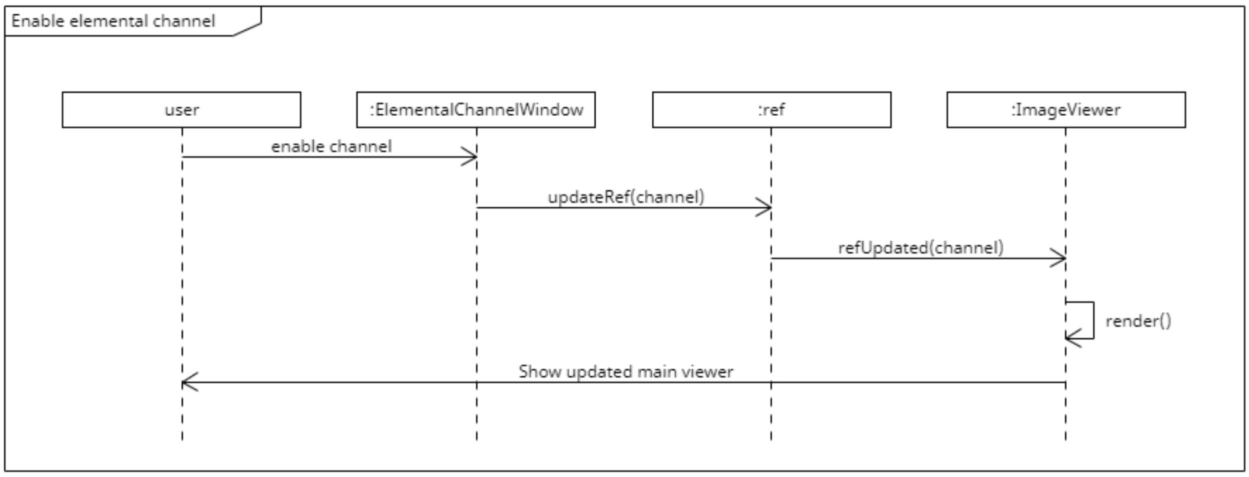

Figure 46: Sequence diagram for enabling an elemental channel

3.3.33 Changing elemental channel highlight color

To change the highlight color of an element, the user has to pick the color selection button, which prompts a colorpicker to open. The user is then able to choose their color, which will update the main viewer.

Goal	To change the highlighting color for an element, allowing for better elemental exploration
Preconditions	Elemental data cube available, and elemental channel window enabled. The elemental layer is enabled in the layer system and the element of which the user is trying to update the color, is enabled.
Postconditions	The highlight color for the selected element is updated in the main viewer.
User	All
Priority	Must Have

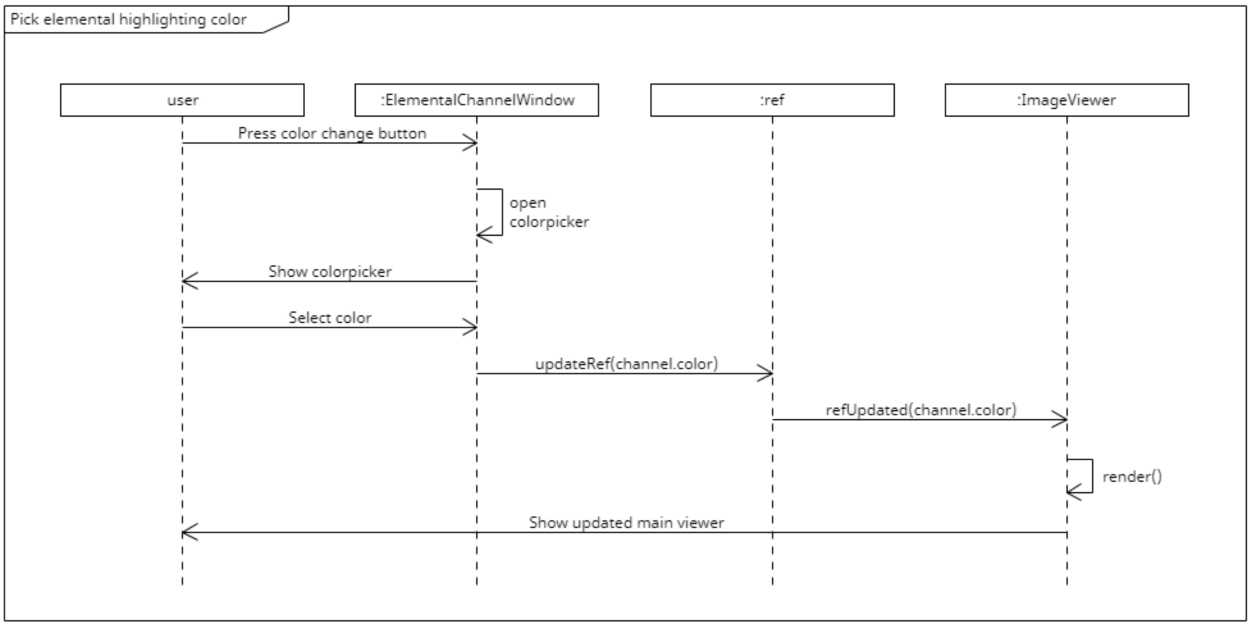

Figure 47: Sequence diagram for changing the highlight color for an elemental channel.

3.3.34 Changing elemental channel highlight intensity

To change the highlight intensity of an element, the user has to use the sliders to calibrate a new highlight intensity threshold. The system then updates the highlight accordingly.

Goal	To change the highlighting intensity for an element, allowing for better elemental exploration
Preconditions	Elemental data cube available, and elemental channel window enabled. The elemental layer is enabled in the layer system and the element of which the user is trying to update the highlight intensity is enabled.
Postconditions	The highlight intensity for the selected element is updated in the main viewer.
User	All
Priority	Must Have

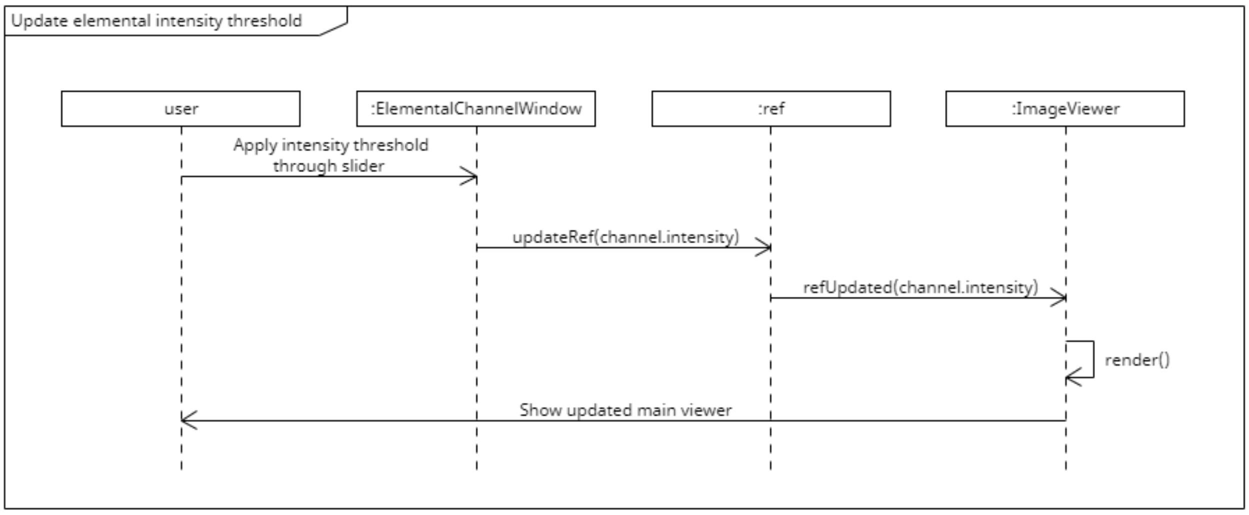
Update elemental intensity threshold
user
:ElementalChannelWindow
:ref
:ImageViewer
Apply intensity threshold through slider
updateRef(channel.intensity)
refUpdated(channel.intensity)
render()
Show updated main viewer

Figure 48: Sequence diagram for changing the highlight intensity for an elemental channel.

3.4 Data model

To be able to use XRF Explorer 2.0, the user needs to a create data source or load an existing one.

By creating or loading at least one valid data source, the user may perform every feature of the application.

Data sources are stored using the folder structure described in Figure 49. Every time a data source is created a new subfolder with that data source name is also created.

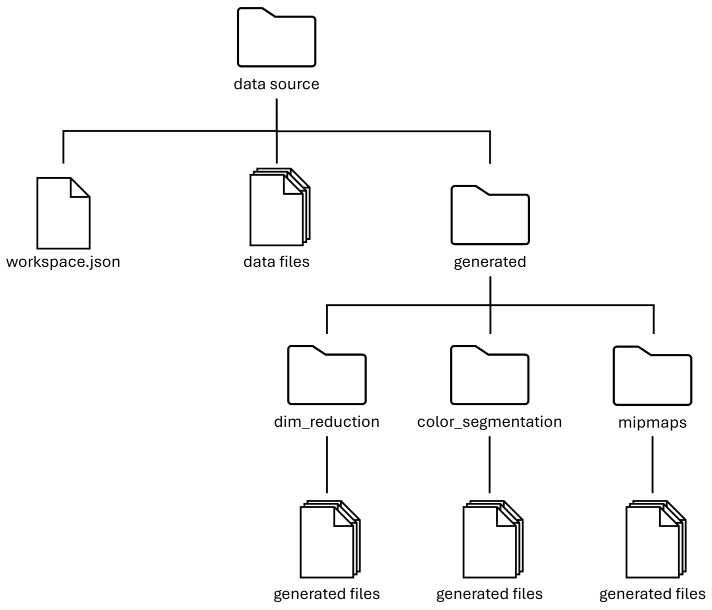

Figure 49: File structure of a data source

Inside the subfolder, there is a subdivision between data files related to the painting (e.g., RGB image, UV image, datacubes.), a `workspace.json` file and a `generated` folder.

The painting's data files and the `workspace.json` file are created when the data source is created. The `generated` folder and its sub-directories are lazily created, i.e. only whenever they are required to store back-end data.

The embedding file is stored in the `dim_reduction` folder. The files stored in the `mipmaps` folder are used to speed up the computation of the spectral data averages. Finally, the `color_segmentation` folder stores the color segmentation bitmasks and JSON files containing the color names.

3.5 External interface definitions

In this section, we describe the external interfaces of the XRF Explorer 2.0. The XRF Explorer 2.0 interacts with both the user's file system and web-browser. The application interacts with the user's file system to both upload and export files. Additionally, the application interacts with the user's browser for the color picker, uploads and downloads, and to display client-side logic.

3.5.1 Interaction with the user's file system:

To upload files, users interact with their local file explorer to select and upload desired files. Similarly, exporting files requires the user's file system to save the main viewer's images and/or visualization views. These images are exported as a PNG to the user's standard download location.

3.5.2 Interaction with the user's browser:

The application interacts with the user's browser in the following ways:

- **Color Picker**: The color picker tool within the application utilizes the browser's built-in color picker interface to allow users to choose colors for highlights.
- **Uploads and Downloads**: The upload and download processes are managed through the browser's file handling. When users upload files, they use the browser's file dialog to navigate their file system. Downloads, such as exporting images, are handled by the browser, saving files to the user's designated download location.
- **Client-side code:** The browser will run the compiled client-side code to display the application and render the images, as well as let the user interact with the application.

3.5.3 Interaction with the server's file system

Uploaded files are stored on the server, such that they can be parsed and used for the more complex computations. When the user uploads a file, it is stored at a pre-configured location in the server, where it can then access the information needed. The server will also generate files, which will also be stored in a pre-configured location on the server.

3.6 Design rationale

In determining an appropriate technology stack to use for XRF Explorer 2.0, we were constrained by three main factors. The requirement to create a web application served by a back-end on a remote server or local computer instead of a typical GUI application, familiarity with the language amongst the team, and familiarity with the tech stack by the client and pre-existing libraries.

For the web app, there were two main design rationales that could be chosen. Server-side rendering (SSR), where all the computation would be offloaded to the back-end, and client-side rendering (CSR), where the back-end only serves the client, with the browser taking care of all computations required for drawing the client. While server-side rendering would allow for closer integration with the back-end server and the computer, the nature of the XRF Explorer 2.0 requires that the client is able to update and re-render itself as efficiently as possible. This ruled out the option to use SSR. In addition, the used data files can be up to 10 GB in size. Robustly handling files of this size is not achievable within a web browser, as access to many OS features, such as memory maps, are heavily locked down and sand-boxed by all modern browsers. This ruled out a purely CSR-driven approach, and we decided on a more hybrid approach.

While the web application itself is purely rendered on the client-side, the back-end server would, in addition to serving the client, also be responsible for all computations not directly related to rendering the front-end through an API. The only remaining choice was to decide on what technologies to use for the front-end and back-end respectively.

3.6.1 Vue.js

As the front-end is rendered on the client-side, using a JS-based framework is required for the client. Additionally, as the client would need to be served without significant overhead from a back-end server, a technology such as Vite or Webpack was required which would compile the front-end source code into a single HTML, JS and CSS file, that could be statically served. For this front-end compilation we chose to use Vite due to positive prior experience with the tool within the team.

As there are many UI frameworks available for developing a front-end, with most of those frameworks built around the same rendering concepts, such as reactivity or signals, the most significant factor for choosing a UI framework was estimated ease of development. Due to the simple layout and syntax of Vue.js' single file components and its stellar integration with Vite, which is developed by the same developers as Vue.js, the decision was made to use Vue.js for the front-end.

Due to the anticipated complexity of the front-end, it was decided early on that it would be important to remove any ambiguity where possible in the frontend codebase. For that reason it was decided that Vue would be used in combination with TypeScript instead of JavaScript, as TypeScript would enforce a strict type system in the code base.

3.6.2 Image Viewer

One of the main components of the front-end is an image viewer that displays different data visualizations, including the painting, color segmentation selection, dimensionality reduction selection, elemental maps and contextual images. For implementing this image viewer, there were two main possibilities.

Similar to the first version of the XRF Explorer it would be possible to modify an existing image viewer such as OpenSeaDragon[4] to work for our use case. The other possible path would be to develop a custom image viewer made to our specific requirements. While using an existing image viewer would lessen the workload, complex image manipulation requirements prevent our use of existing image viewers.

Most image viewers are not capable of handling images the size of the largest images that the XRF Explorer 2.0 is required to support, and those that do, have no or limited support for handling multiple images simultaneously, with none of the investigated image viewers having support for registering and displaying multiple images of different sizes. Hence, the decision was made to utilize THREE.js to create a custom image viewer, as using the WebGL ecosystem of vertex and fragment shaders in combination with the scene management of THREE.js would allow us to create a custom image viewer at a relatively low complexity while providing full control in customizing every step of the rendering process.

Image viewer	8192x8192 images	Multiple images	Registering images
Viewer.js [5]	no	no	no
IIPMooViewer [6]	yes	no	no
OpenSeaDragon [4]	yes	with modifications	no
Custom renderer	yes	yes	yes

3.6.3 D3.js

In addition to the image viewer in the frontend, the application is also required to display auxiliary data such as elemental compositions and spectral data as charts. As previous work used D3.js to draw these graphs and some team members had positive prior experience with the library, the decision was made to use D3.js to draw these charts.

3.6.4 Python

In choosing a language for the back-end of the application, a multitude of factors was considered. First and foremost, given the short time frame of the project, we had to choose a language all, or at least a large majority of the team members were already familiar with. In addition, the choice for the back-end is dictated by existing libraries necessary for the required data processing for the XRF Explorer 2.0. As the previous work is based on a Python back-end, Python has an extensive array of data processing libraries, specifically numpy and umap-learn, most team members had pre-existing familiarity with Python, and the client's previous projects were already based on a Python codebase, the decision was made to use Python. An alternative to Python could have been to use JavaScript for the backend using Node.js, however, this was decided against due to the team members being more familiar with Python and the lack of available libraries for dimensionality reduction.

3.6.5 Flask

For creating the web server on the back-end to serve both the client and the associated API, a choice in Python web framework had to be made. The most prevalent of these frameworks are Django and Flask. Due to the relative simplicity of Flask compared to Django, especially due to how Flask handles creating API routes through function annotations, it was decided to use Flask over Django. In addition, the previous work also made use of Flask to serve the client, giving us the confidence that Flask would be able to handle all of the requests required to implement all parts of the XRF Explorer 2.0.

4 FEASIBILITY AND RESOURCE ESTIMATES

This section provides a summary of the computer resources required to build, operate, and maintain the XRF Explorer 2.0 software.

4.1 Hardware requirements

4.1.1 Hardware testing

To get a better estimate of the hardware requirements, performance tests were done. The following machine was used:

Machine	Lenovo ThinkPad P1 (3rd generation)
CPU	i7-10750H, 6 cores @2.6GHz
Operating System	Windows 11 (64-bit)
Memory	16GB @3.200MT/s
Disk space	512GB SSD

During the test session, all applications except for the task manager, two terminals for running the back-end and front-end and a web browser, were terminated. The web application's performance was tested using Firefox (version 125) and Google Chrome (version 124), but performance differences were negligible.

Results and recommendations

For the majority of the test session, the memory usage percentage hovered between 50% and 60%, with short peaks around 75% while doing more involved computations. The CPU usage remained at 25%, though while generating the dimensionality reduction embedding it increased to 80% and occasionally reached peaks of 100%.

Based on these results, a minimum of 16GB of RAM is recommended. While it might theoretically be possible to run the application using only 8GB of RAM, in a more practical scenario this is not feasible. A strong CPU recommendation is harder to formulate. The CPU sat comfortably at 25% for large portions of the test session but struggled while doing more complex calculations. Considering that the intended usage of the web application involves regularly recalculating embedding and color clusters, the peak results should not be ignored. The CPU used for testing struggled at moments, but this was not noticeable to the user. As the application is capable of doing multiple calculations simultaneously, the recommendation is to increase the number of cores when compared to the testing machine (as opposed to increasing the speed).

Concerning disk space and bandwidth, the disk space is largely dependent on the number of data sources the user wants to store. The raw data can be around 10GB and, as such, if the user wants to have multiple spectral cubes per data source or multiple data sources, the disk space should be scaled accordingly. For running the application locally, 512GB is appropriate (taking into account that the user won't use the device only for this application), but 256GB would also suffice. The server, where multiple users might want to upload data sources and access them, should have a large storage space and, as such, 1TB of disk space is recommended. The bandwidth of the server should be high enough that uploading 10GB files is feasible. Our recommendation is a server with a bandwidth of $\sim$100Mbps. At this speed, uploading the largest files would still take around 15 minutes, but thereafter the bandwidth is no longer a huge factor. $\sim$100 Mbps would thus be an acceptable trade-off between upload times for the user's data sources and the costs of higher bandwidths.

Furthermore, it should be noted that a graphics card is required as the application makes use of WebGL. Any modern-day graphics card will suffice, including CPU-integrated models.

4.1.2 Server requirements

Server requirements: Windows

CPU	Intel Core i9-10900F, 10 cores @2.8GHz
Operating System	Windows 10 (64-bit), Windows 11 (64-bit)
Memory	16 GB
Disk space	1TB
Network	100 Mbps
Browser	-
Other Software	Chocolatey, Git, Python, npm

Server requirements: Linux

CPU	Intel Core i9-10900F, 10 cores @2.8GHz
Operating System	Ubuntu LTS (version 24.04 or higher)
Memory	16 GB
Disk space	1TB
Network	100 Mbps
Browser	-
Other Software	Git, Python, npm

4.1.3 Client machine requirements

These requirements are for running the application locally and not for accessing the webserver.

Client requirements: Windows

CPU	Intel Core i9-10900F, 10 cores @2.8GHz
Operating System	Windows 10 (64-bit), Windows 11 (64-bit)
Memory	16 GB
Disk space	512GB
Network	Not relevant while running locally
Browser	Google Chrome (version 124 or higher), or Firefox (version 125 or higher)
Other Software	Python, npm

Client requirements: Linux

CPU	Intel Core i9-10900F, 10 cores @2.8GHz
Operating System	Ubuntu Desktop (version 22.04 or higher)
Memory	16 GB
Disk space	512GB
Network	Not relevant while running locally
Browser	Google Chrome (version 124 or higher), or Firefox (version 125 or higher)
Other Software	Python, npm

4.1.4 Software requirements

Python	3.12
npm	10.5.0

4.2 Performance metrics

This section gives an overview of the performance of the XRF Explorer 2.0 when carrying out various tasks. For each task, it gives an overview of the different steps needed to complete it, as well as the runtime taken for each subtask as well as the task as a whole.

The performance tests were run on the server provided by the clients, through browser on the laptop of one of the team members. The server uses 64-bit 32 core AMD EPYC 7542 processor with 32GB (16x2) of RAM running Ubuntu 24.04 LTS. The laptop used is the same as the one described at the start of Section 4.1.1. using Firefox version 127.

The data used for the performance tests is the data provided by the clients, which includes a 10GB spectral data file, an elemental data file, a high-quality RGB image, a UV image, and an X-ray image.

4.2.1 Project creation

This task includes creating a new project. The total time is measured from when the user clicks the ”Save” button in the ”Set up workspace data” dialog.

Subtask	Subtask time	Total time
Save uploaded data	100ms	100ms
Load elements	11ms	111ms

4.2.2 Project startup

This task includes loading a project for the first time, and binning the spectral data (done in the background). The total time is measured from when the user loads the project.

Subtask	Subtask time	Total time
Load workspace	831ms	831ms
Serve RGB image	68ms	899ms
Bin data	66 622ms	67 521ms

4.2.3 Spectral charts global average

This task includes computing the spectral global average and displaying it on the spectral charts. The total time is measured from when the user opens the spectral chart window.

Subtask	Subtask time	Total time
Get selection at respective mip level	405ms	405ms
Calculating average within selection	13ms	418ms
Selection average plotted and displayed to user	506ms	924ms

4.2.4 Spectral charts selection average

This task includes computing the spectral selection average and displaying it on the spectral charts. The total time is measured from when the user makes the selection.

Subtask	Subtask time	Total time
Get selection at respective mip level	582ms	582ms
Calculating average within selection	13ms	595ms
Selection average plotted and displayed to user	345ms	940ms

4.2.5 Elemental charts global average

This task includes computing the elemental global average and displaying it on the elemental charts: The total time is measured from when the user opens the elemental chart window.

Subtask	Subtask time	Total time
Read elemental data cube	302ms	302ms
Load elemental data cube	213ms	515ms
Load elements	0ms	515ms
Compute average within selection	210ms	725ms
Selection average plotted and displayed to user	580ms	1 305ms

4.2.6 Elemental charts selection average

This task includes computing the elemental selection average and displaying it on the elemental charts. The total time is measured from when the user makes the selection.

Subtask	Subtask time	Total time
Read elemental data cube	378ms	378ms
Load elemental data cube	51ms	429ms
Load elements	0ms	429ms
Compute average within selection	711ms	1 140ms
Selection average plotted and displayed to user	650ms	1 790ms

4.2.7 Dimensionality reduction

This task includes generating an embedding, generating the respective overlay, and displaying it to the user. The total time is measured from when the user clicks the ”Generate embedding” button.

Subtask	Subtask time	Total time
Read elemental data cube	318ms	318ms
Load elemental data cube	51ms	369ms
Generate embedding	114 153ms	114 533ms
Generate overlay	1 363ms	115 516ms
Generate embedding image	2 075ms	117 591ms
Image displayed to user	740ms	118 331ms

4.2.8 Color segmentation

This task includes computing the color clusters, saving the corresponding bitmask, and displaying it to the user. The total time is measured from when the user clicks the ”Generate color clusters” button.

Subtask	Subtask time	Total time
Read elemental data cube	220ms	220ms
Load elemental data cube	73ms	293ms
Compute color clusters	593ms	832ms
Combine and save bitmasks	360ms	1 192ms
Display clusters to user	472ms	1 664ms

A FRONT-END CLASS DIAGRAM

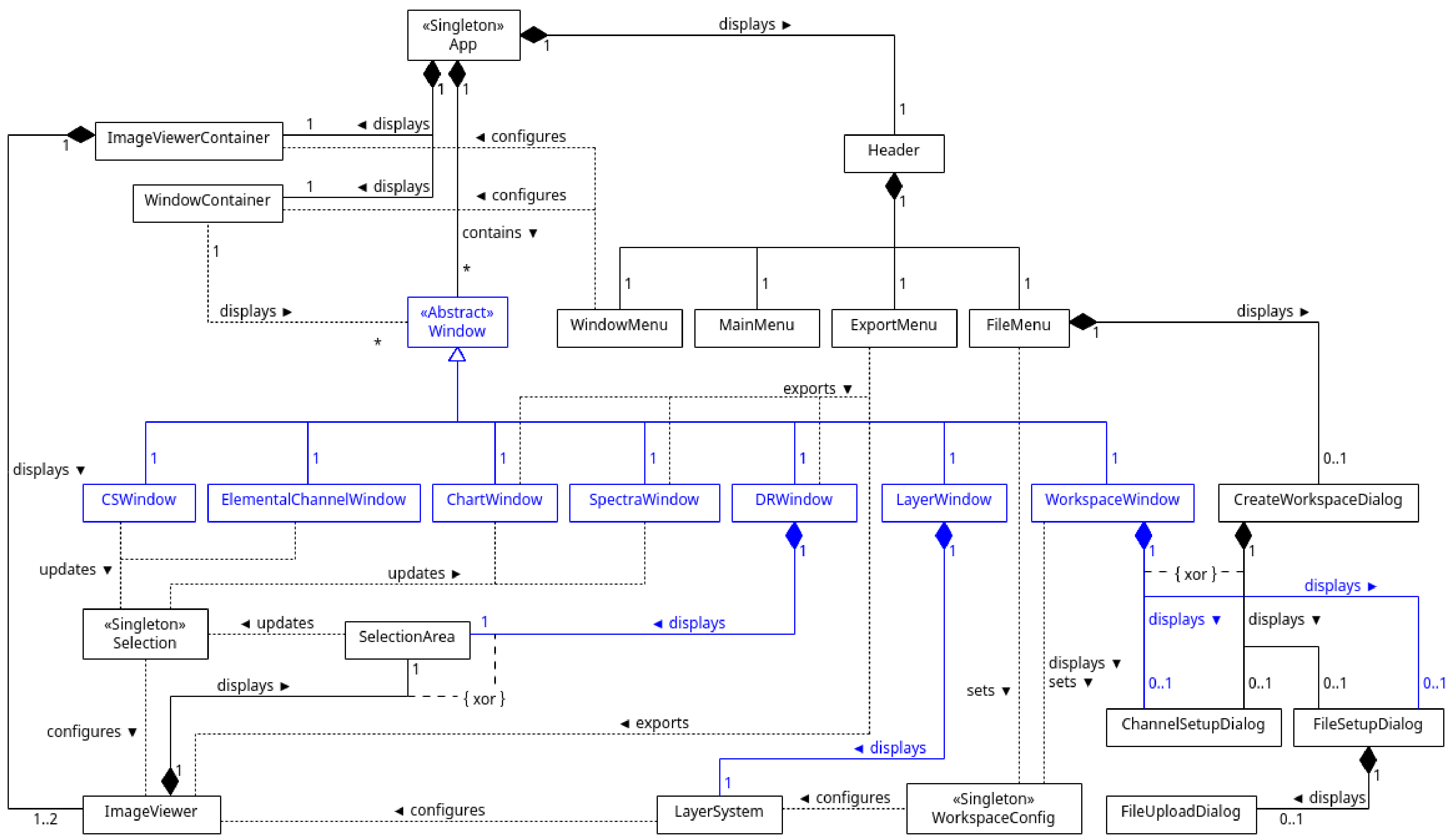

Figure 50: Logical model of the front-end