ISO/IEC JTC 1/SC 29/WG1M96014
96th meeting – Online – July 25-29, 2022

ISO/IEC JTC 1/SC 29/WG 1
(ITU-T SG16)

Coding of Still Pictures

JBIG	**JPEG**
Joint Bi-level Image Experts Group	Joint Photographic Experts Group

TITLE: UPC Proposal for JUMBF (ISO/IEC 19566-5) Ed2 Reference Software

SOURCE: Nikolaos Fotos, Jaime Delgado (DMAG/IMP-UPC)
Distributed Multimedia Applications Group (DMAG) /
Information Modeling and Processing Research Group (IMP)
Universitat Politècnica de Catalunya – UPC BarcelonaTECH
jaime.delgado@upc.edu, nikolaos.fotos@estudiantat.upc.edu

PROJECT: ISO/IEC 19566-5
(JPEG Systems - Part 5: JPEG universal metadata box format (JUMBF))

STATUS: Proposal

REQUESTED ACTION: Input contribution

DISTRIBUTION: WG1

UPC proposal for
JUMBF (ISO/IEC 19566-5) Ed2 Reference Software

Nikolaos Fotos, Jaime Delgado
UPC BarcelonaTECH

1. Introduction

There are different software tools available for handling JPEG files including JUMBF content. However, no Reference Software is still available. This document presents a new proposal for Reference Software associated to ISO/IEC 19566-5; i.e. JPEG Systems - Part 5: JPEG universal metadata box format (JUMBF). The first proposal (document WG1/m95048) implemented the first edition of the standard part, while the current proposal supports the 2nd version, currently under DIS ballot.

The presented software is called jumbf-core-2.0, it is part of the mipams-jumbf project and it is a library written in Java that allows parsing and generating independent JUMBF files. In addition, this second version of the software implements the embedding of JUMBF structures inside JPEG images. The design of this software aims to be extended in order to support JUMBF structures from other JPEG Systems standards. A first application of this software was presented in document ISO/IEC JTC1 SC29/WG1/m95049: “UPC proposal for JPEG Systems’ Privacy & Security (ISO/IEC 19566-4) Reference Software: An application of the proposed JUMBF Reference Software”. An update of that document, numbered ISO/IEC JTC1 SC29/WG1/M96015, has been submitted.

This document provides background information on the software design approach followed for the proposed reference software for JPEG Universal Metadata Box Format (JUMBF), as defined in JPEG Systems Part 5 standard: ISO/IEC DIS 19566-5:2022 Information technologies - JPEG systems - Part 5: JPEG universal metadata box format (JUMBF).

Section 2 provides high-level information of the functionalities that the proposed reference software supports. Next, Section 3 presents the hierarchical structure and the dependencies of the classes that are used in the software. Subsequently, Section 4 presents a use case that demonstrates how JUMBF structure is generated based on the classes defined in Section 3. Finally, in Section 5, we demonstrate an example application that uses our library and provides a graphical user interface (GUI) that allows a user to generate and parse metadata in JUMB Format using.

The software is available at https:www.github.com/nickft/mipams-jumbf

2. Functionalities of jumbf-core library

The jumbf-core-2.0 library provides a way to generate and parse information that is stored in JUMB Format. Furthermore, it defines an interface so that developers can extend its functionality and support additional JUMBF Box structures as specified in other JPEG standards (e.g. Protection Box definition in JPEG Systems Part 4: Privacy & Security standard).

In the first version of the library (i.e. jumbf-core-1.0), we embedded JUMBF structure information to a file using the ISO Base Media File Format (ISOBMFF). Thus, we specify the file extension ".jumbf" that contains a list (i.e. concatenation) of ISOBMFF boxes. Each box corresponds to one JUMBF box. This allows us to store the metadata information separately from the media asset itself.

Storing JUMBF metadata in a separate ".jumbf" file is not yet standardized. Thus, it has been decided that from jumbf-core-2.0 JUMBF metadata need to be also embedded in a digital asset as well. For this reason, two additional services have been defined in the second version of jumbf core, namely JpegCodestreamGenerator and JpegCodestreamParser. With these two services it is possible to parse and embed JUMBF metadata inside a JPEG XT and JPEG 1 encoded images. For the scope of this Project, it is necessary to handle JUMBF metadata inside the image; all other information is simply skipped during the parsing/generation procedures.

In the following section we present how we implement the boxes that are defined in the scope of the JPEG Systems Part 5 standard.

3. Software Design: Class Inheritance

In this section we specify the classes that have been designed in order to support the data model presented in ISO/IEC 19566-5 standard. The core concept in this data model is the Box.

To describe a Box structure in jumbf-core library two classes need to be specified: an Entity class and a Service class. An Entity class contains the information regarding the fields that are defined in the specific Box. In addition, all the functionality which is required to parse/generate a particular box is included in its respective Service class. This allows for a better separation of concerns in our software.

Furthermore, once we have defined the necessary Box structures we need to take into account the different Content Types of a JUMBF Box. The number as well as the type of the Content Boxes inside a JUMBF Box is specified from the Content Type UUID which is located in its Description box fields. Consequently, an additional Content Type class hierarchy. Each Content Type class provides the means to parse and generate the necessary Content Boxes inside a JUMBF Box.

3.1 Box structure: Entity and Service classes

As it is mentioned in the Introduction, when we are referring to a Box structure we assume that it is wrapped with the ISOBMFF headers. In other words, before writing a Box structure to a file, we need to reserve a set of bytes in order to write the ISOBMFF headers, namely:

- The Length of the box (LBox): 4 Bytes
- The Type of the box (TBox): 4 Bytes
- The box length extension (XLBox): 8 Bytes

With that in mind, we are ready to present the two class hierarchies that are required to specify a box structure: Entity and Service class hierarchies. In Figure 1, we see the entire Entity class hierarchy as defined in jumbf-core-2.0.

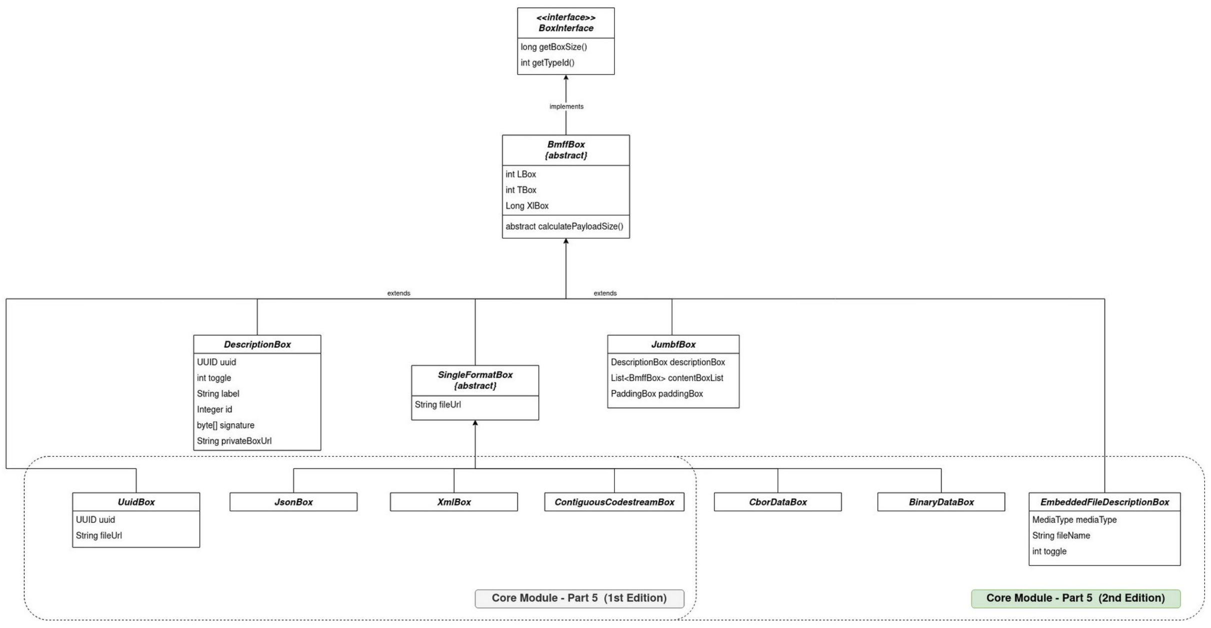


Figure 1: Entity class hierarchy in jumbf-core-2.0

The core of the Entity class hierarchy is the BoxInterface (at the top of Figure 1) interface which defines two methods “Get the Box Type Id” and “Get the size of the Box”. In the next level we see a BmffBox abstract class that defines the fields which correspond to the ISOBMFF header fields. Subsequently, we can see the Description box definition (e.g. DescriptionBox class) along with the fields that are defined in the first edition of ISO/IEC 19566-5. With these definitions, we are ready to implement the Entity class for a JUMBF Box as a set of fields consisting of exactly one DescriptionBox field and a list of BmffBox fields corresponding to the Content Boxes according to the Content Type specified in DescriptionBox’s uuid field. Finally, in Figure 1, all the Content Box definitions are listed along with their respective fields. At the bottom level of the figure, we can see the box structures defined in 1st edition of ISO/IEC 19566-5 as well as the ones that are defined in the draft versions of 2nd edition.

Note: The validity of the content of the content boxes is out of scope of the jumbf-core-2.0. It is up to the application that uses this library to evaluate the content of a Content Box.
Thus, we have defined an abstract class SingleFormatBox that contains a single "fileUrl" field which contains the URL to the file containing the contents that should be included in a Content Box. As depicted in Figure 1, this abstract class is extended by the following Content Boxes: JSON, XML, Contiguous Codestream (JP2C), CBOR and Binary Data Content boxes.

Now that an Entity class is defined for each Box we need to specify the methods to parse and generate it. Thus, we need to define the respective Service class for each Box. Each Service should implement the methods to parse and generate only the fields that are defined in its corresponding Box. In Figure 2 the entire Service class hierarchy is depicted.

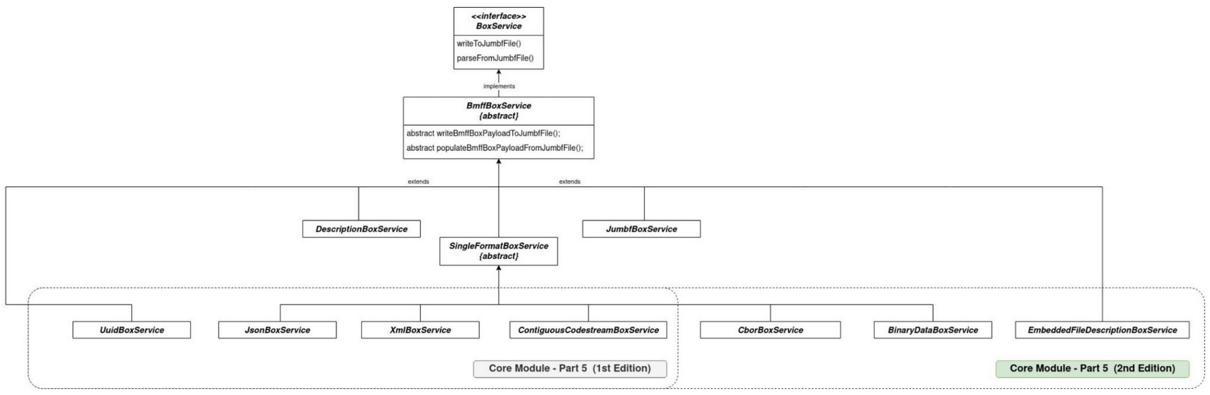


Figure 2: Service class hierarchy

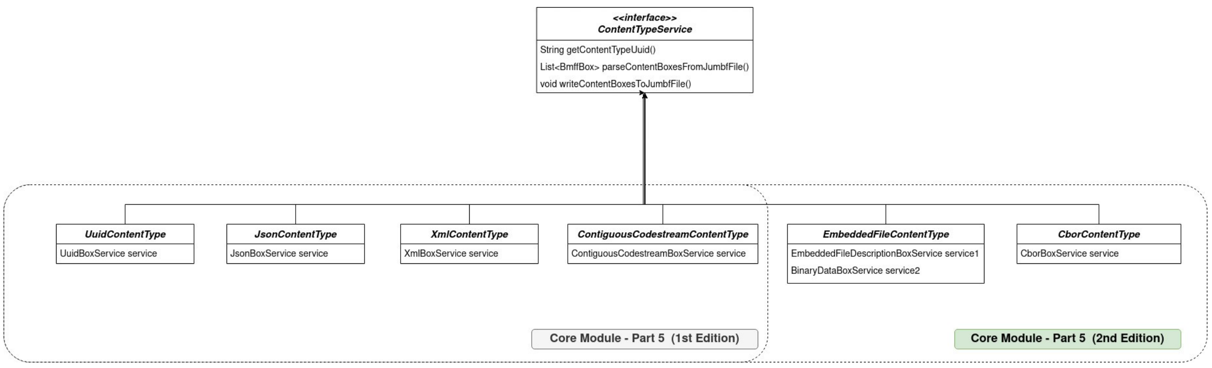
The structure seems similar to the Entity class hierarchy. Specifically, at the top level we define an Interface, namely BoxService, that specifies two methods: one for writing a Box Entity to a JUMBF file and one for instantiating a Box Entity by parsing a JUMBF file. Then, we define the BmffBoxService abstract class which contains the functionality to write an ISOBMFF structure to a file. It is evident that at this level the BmffBoxService class knows only how to handle the ISOBMFF headers of an Entity class. The way to handle the ISOBMFF payload shall be specified by the corresponding BoxService class that extends this BmffBoxService class. This is the reason BmffBoxService class marks the methods "Write ISOBMFF Box Payload" and "Populate Bmff Box Payload" as abstract classes. In the remaining levels we see all the Services that specify how to handle each Box that we defined in the Entity class hierarchy.

3.2 Content Type classes

In Figure 1, we defined the JUMBF box entity class as consisting of exactly one Description box and a set of ISOBMFF Boxes corresponding to the Content Boxes. In addition, in Figure 2 we explained that the JumbfBoxService class provides the means to handle the fields of a JUMBF Box. Although the DescriptionBoxService is always called when implementing a JumbfBoxService, the set of BoxService classes that are needed depends on the Content Type of the JUMBF Box which is located in the Description Box Entity "uuid" field.

For instance, for a JSON Content type JUMBF box the JumbfBoxService class will invoke - apart from the DescriptionBoxService - the JsonBoxService class. However, this is not the case for an Embedded File Content type JUMBF box which consists of two Content Boxes, namely an Embedded File Description Box and a Binary Data Box. In this example, the JumbfBoxService class will call the EmbeddedFileDescriptionBoxService class as well as the BinaryDataBoxService class.

Consequently, in addition to the Boxes hierarchies, it is required to define a family of Content Type classes that provide the means to handle the set of Content Boxes that are required according to the Content Type UUID. The Content Type hierarchy is depicted in Figure 3.

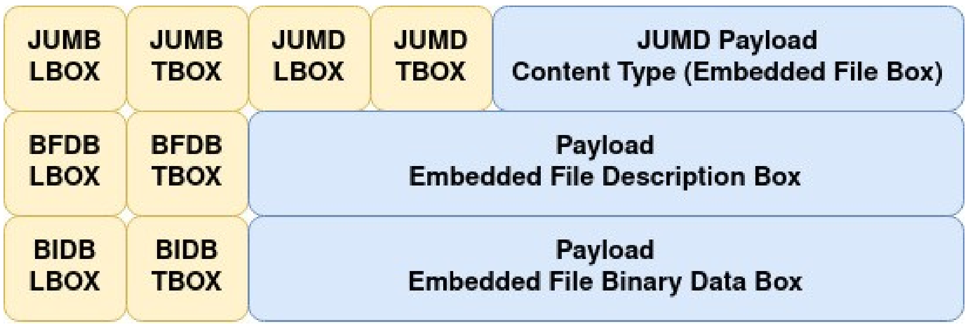

<<interface>>
ContentTypeService
String getContentTypeUuid()
List<BmffBox> parseContentBoxesFromJumbfFile()
void writeContentBoxesToJumbfFile()
UuidContentType
UuidBoxService service
JsonContentType
JsonBoxService service
XmlContentType
XmlBoxService service
ContiguousCodestreamContentType
ContiguousCodestreamBoxService service
EmbeddedFileContentType
EmbeddedFileDescriptionBoxService service1
BinaryDataBoxService service2
CborContentType
CborBoxService service
Core Module - Part 5 (1st Edition)
Core Module - Part 5 (2nd Edition)
Figure 3: Content Type class hierarchy

At the top level of Figure 3, there is the core ContentTypeService interface which defines three important functionalities that each Content Type class shall implement: “Get the Content Type UUID”, “Parse the JUMBF Content Boxes from the JUMBF file” and “write the JUMBF Content Boxes to a JUMBF file”.

At the second level of Figure 3, all the Content Types supported in jumbf-core-2.0 are displayed. In each of the ContentType services you can find the respective BoxService classe(s) required to handle the necessary Content Boxes. As we mentioned before, according to ISO/IEC 19566-5 standard only the Embedded File Content type JUMBF box is the one that requires more than one Content Box.

4. Example: Generating a JUMBF file

In this section we will demonstrate the invocations made jumbf-core-2.0 during the generation of a JUMBF file (i.e. the output is a file with .jumbf extension). First and foremost, we assume that we have already built an instance of a JumbfBox Entity class that describes an Embedded File Content type JUMBF box. The output byte stream is logically grouped in Figure 4. We can clearly see the bytes that correspond to the ISOBMFF headers as well as those bytes that correspond to the Boxes described in the previous section.


JUMB LBOX
JUMB TBOX
JUMD LBOX
JUMD TBOX
JUMD Payload Content Type (Embedded File Box)
BFDB LBOX
BFDB TBOX
Payload Embedded File Description Box
BIDB LBOX
BIDB TBOX
Payload Embedded File Binary Data Box
Figure 4: Embedded File Content type JUMBF box stored in a JUMBF file.

Note: The orange boxes in Figure 4 correspond to sets of exactly 4 bytes while the blue boxes are of variable length. The label of each box contains the Box Type that these bytes are referring to as well as the exact content of each box. For example the first box corresponds to the ISOBMFF LBox header that refers to the JUMBF box (i.e. the JUMBF box type is 'jumb' or 0x6A756D62).

The algorithm to generate a JUMBF file begins from the JUMBFBoxService class:

A. First, **JumbfBoxService** writes the ISOBMFF header bytes (JUMB LBox and TBox) through the functionality extended by the BmffBoxService class.
B. **JumbfBoxService** calls the DescriptionBoxService to proceed with its box byte generation.
C. **DescriptionBoxService** writes the ISOBMFF header bytes (JUMD LBox and TBox) through the functionality extended by the BmffBoxService class.
D. **DescriptionBoxService** writes its fields (e.g. Content Type UUID, Toggle and other optional fields). The operation returns back to the JumbfBoxService.
E. **JumbfBoxService** reads the Content Type UUID from the Description Box and chooses the corresponding BoxService to proceed with the JUMBF Content Box byte generation. In this example the correct service is the EmbeddedFileContentType.
F. **EmbeddedFileContentType** calls the EmbeddedFileDescriptionBoxService to proceed with its box byte generation.
G. **EmbeddedFileDescriptionBoxService** writes the ISOBMFF header bytes (BFDB LBox and TBox) through the functionality extended by the BmffBoxService class.
H. **EmbeddedFileDescriptionBoxService** writes its fields (e.g. Content Type UUID, Toggle and other optional fields). The operation returns back to the **EmbeddedFileContentType**.
I. **EmbeddedFileContentType** calls the BinaryDataBoxService to proceed with its box byte generation.
J. **BinaryDataBoxService** writes the ISOBMFF header bytes (BFDB LBox and TBox) through the functionality extended by the BmffBoxService class.
K. **BinaryDataBoxService** writes the bytes specified in the Binary Data Box structure. The operation returns back to EmbeddedFileContentType.
L. **EmbeddedFileContentType** finishes its operation which returns back to JumbfBoxService.
M. **JumbfBoxService** finishes its operation and the final .jumbf file has been generated successfully.

5. Demo application

In this subsection we present an example application that uses jumbf-core-2.0 library. Specifically, it provides a graphical user interface allowing to parse and generate JUMBF metadata in .jumbf file.

This demo application consists of two services: the RESTful API server that uses the jumbf-core library and the client application that provides the GUI. We have launched these two services using docker containers. For more information on how to run the demo application visit the Github repository attached in the Introduction section. Once both services are up and running we can open a browser and connect to the GUI.

In Figure 5, the main page of the GUI is depicted.

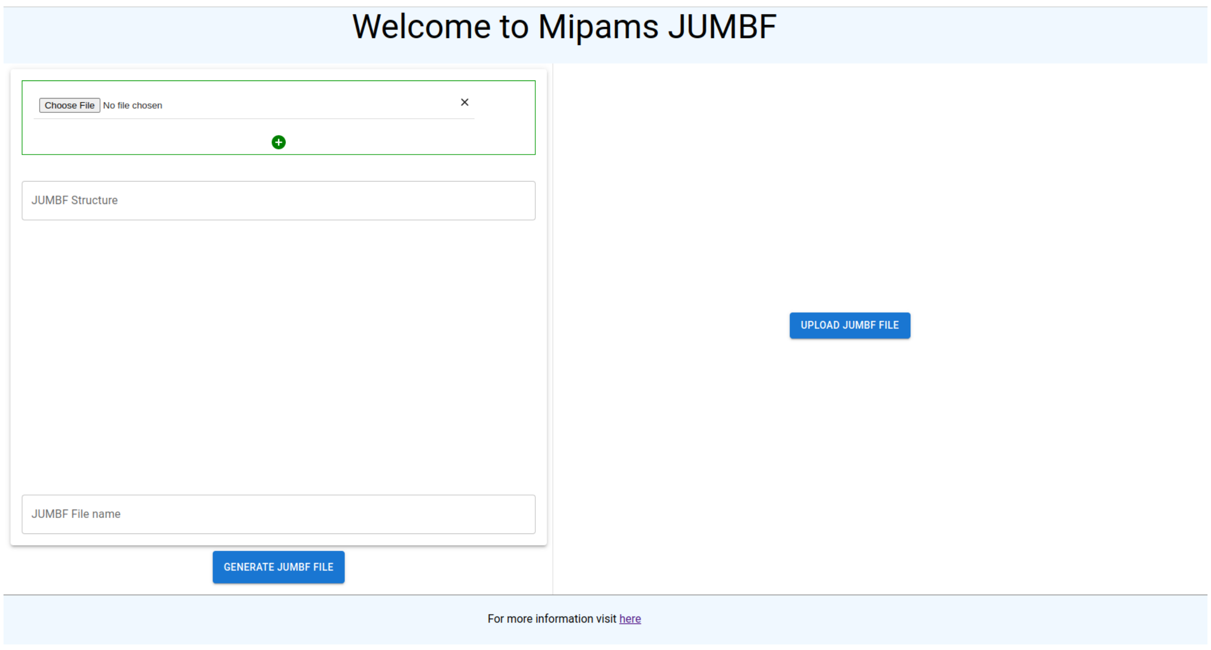
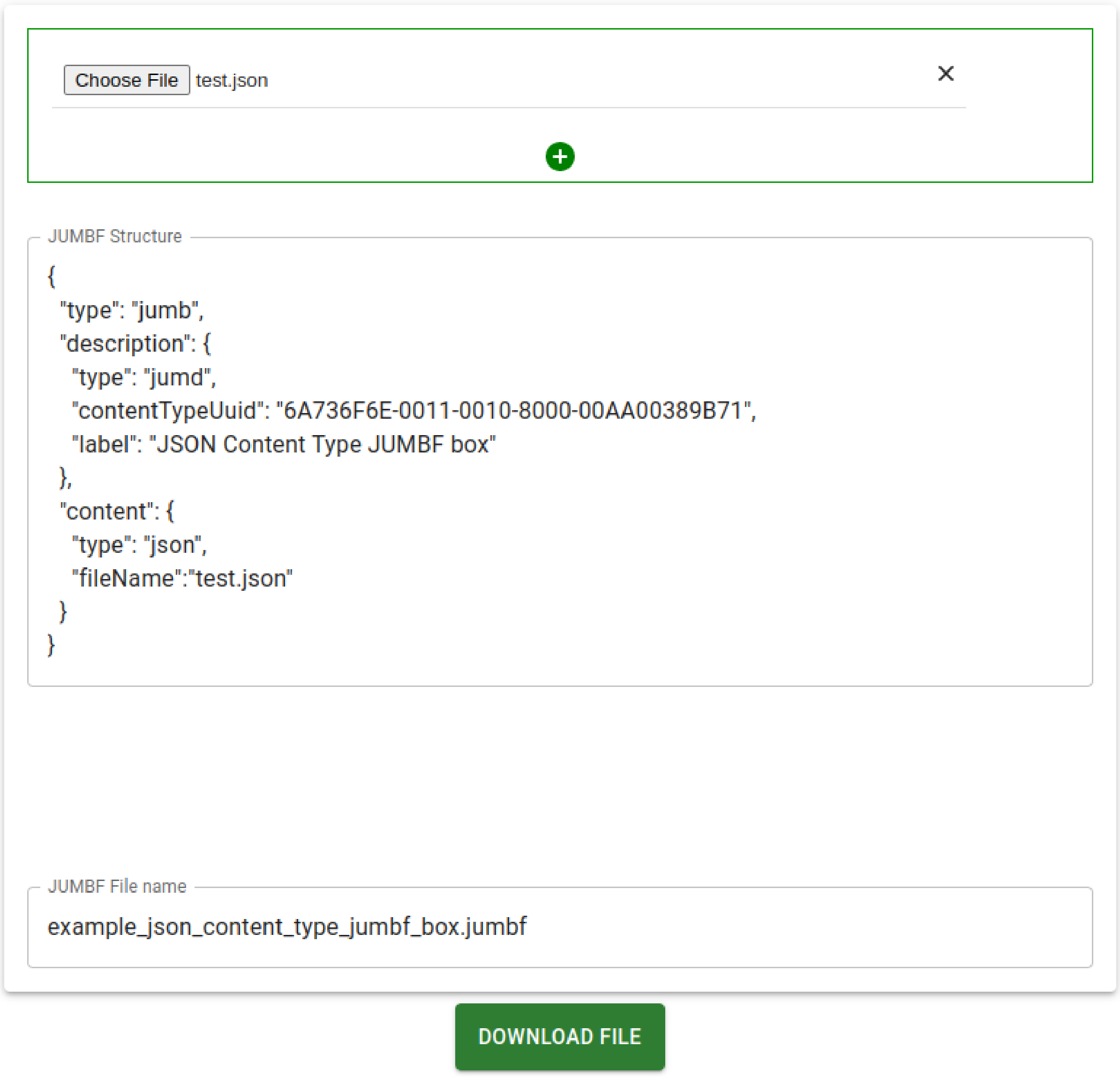


Figure 5: Main page of the application

The GUI is divided into two main components: One the left side, the user interacts to Generate a JUMBF file while on the right side, the user uploads a JUMBF File to inspect its contents.

Firstly, we demonstrate how to generate a JUMBF file. For the scope of this demo application, a simple syntax format has been defined in order for the user to express a JUMBF structure with JSON. An example is shown in Figure 6.



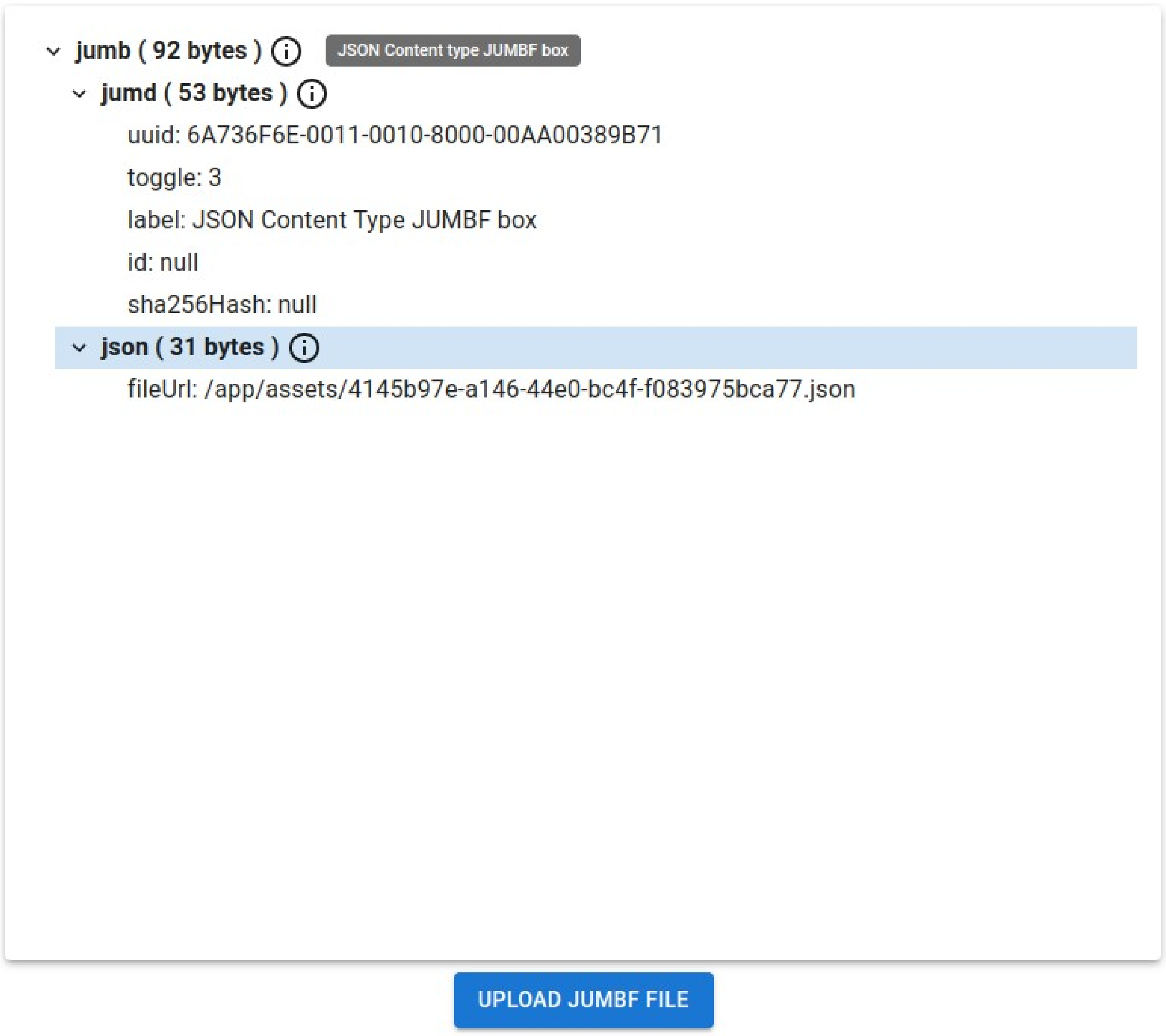
```
{
  "type": "jumb",
  "description": {
    "type": "jumd",
    "contentTypeUuid": "6A736F6E-0011-0010-8000-00AA00389B71",
    "label": "JSON Content Type JUMBF box"
  },
  "content": {
    "type": "json",
    "fileName":"test.json"
  }
}
```

example_json_content_type_jumbf_box.jumbf

Figure 6: Generating a JSON Content type JUMBF box

Notice that the user has initially specified the file that needs to be uploaded to the application containing the metadata in JSON format (i.e. test.json). Subsequently, it uses the application's supported syntax to express a JUMBF Box with Content Type UUID corresponding to the JSON Content type and an example label. According to the "content" attribute, the content box of this JUMBF Box is a JSON box that contains the information located in the uploaded file. Finally, the user specifies the name of the target JUMBF file and clicks on the "Generate JUMBF File". The process has been performed successfully and a green "Download File" button appears, allowing the user to download the generated JUMBF file.

Now that we have generated a JUMBF file, we can provide it as an input to the parser functionality. From Figure 5, we noticed that there is a button at the right panel that says "Generate JUMBF File". By clicking on that button, the user is asked to select a JUMBF file to upload. Upon selection of the JUMBF file, the application parses its structure and provides a hierarchical structure explaining the ISOBMFF box types as well as the fields that they include. In our example the result of the parsing is depicted in Figure 7.



jumb (92 bytes) ⓘ

jumd (53 bytes) ⓘ

uuid: 6A736F6E-0011-0010-8000-00AA00389B71

toggle: 3

label: JSON Content Type JUMBF box

id: null

sha256Hash: null

json (31 bytes) ⓘ

fileUrl: /app/assets/4145b97e-a146-44e0-bc4f-f083975bca77.json

Figure 7: Parsing a JUMBF file

First of all, by reviewing the structure we can see the ISOBMFF type of each box along with its size (expressed in bytes) and an information logo providing a short description for the box. In the case of a JUMBF box this description explains the Content Type of the JUMBF Box while in any other case the description of the ISOBMFF TBox type is provided.

In addition, we can review the supported fields of each Box structure. Finally, we can review the content of the JSON box. Upon parsing the JUMBF File, jumbf-core-.0 library extracts the embedded information (in this case its data in JSON format) and stores it in a separate file with the corresponding format and a random name.

6. Conclusions

This document presents a library to handle (parse and generate) JPEG universal Multimedia Box Format (JUMBF) content which is eventually stored to a file using the ISOBMFF format.

We have seen that, according to the software design, to support a new Box, two classes need to be implemented: an Entity and a Service class. The Entity class defines the fields of a specific Box structure while the Service class specifies the methods to parse and generate these fields from a .jumbf file.

In the second part, we examined the steps on how jumbf-core-2.0 library writes an Embedded File Content Type JUMBF Box to a file by analysing the bytes that each Service class writes. Finally, we presented a demo application that uses jumbf-core-2.0 library and allows a user to parse/generate .jumbf files.

The software presented here could be the basis for JUMBF Reference Software, to be potentially included in the future Part 10 of JPEG Systems.

Acknowledgements

The work presented in this contribution has been partially supported by the Spanish Government under the project GenClinLab-Sec (PID2020-114394RB-C31, funded by MCIN/AEI/10.13039/501100011033) and by the Generalitat de Catalunya (2017 SGR 174).