# Setting a PACS on FHIR

Sébastien Jodogne[a]
*Institute of Information and Communication Technologies, Electronics and Applied Mathematics (ICTEAM),*

Keywords: FHIR, DICOM, Medical Imaging, Free and Open-Source Software.

Abstract: FHIR has quickly emerged as the most important standard for the interoperability of clinical data. FHIR includes support for medical imaging through its `ImagingStudy` resource that can be used to create mappings between FHIR entities and DICOM studies using the DICOMweb protocol. Unfortunately, there is currently a lack of openly available software implementation combining FHIR with DICOM, which hinders the development of new applications linking clinical data with imaging data. In this paper, the Orthanc server for medical imaging is associated with the HAPI framework in a consistent framework to propose an implementation of the `ImagingStudy` FHIR resource that is tightly coupled with the content of a DICOM server. The resulting framework is released as free and open-source software, with the goals of promoting support for medical imaging in FHIR, of sharing technological knowledge about medical interoperability, and of providing a test environment for developing new healthcare-related applications.

## 1 INTRODUCTION

Technical interoperability refers to the ability of different healthcare systems, software, and devices to communicate and exchange data effectively and efficiently (Benson and Grieve, 2021). Technical interoperability is critical for patient-centered care and for continuity of care because patient information is typically spread over multiple software entities, such as electronic health records, hospital information system, laboratory information system, or pharmacy information system, possibly across different hospitals and different general practitioners. Technical interoperability is also becoming increasingly important as a result of the growing interest in telemedicine, in teleexpertise, in patient empowerment, and in artificial intelligence.

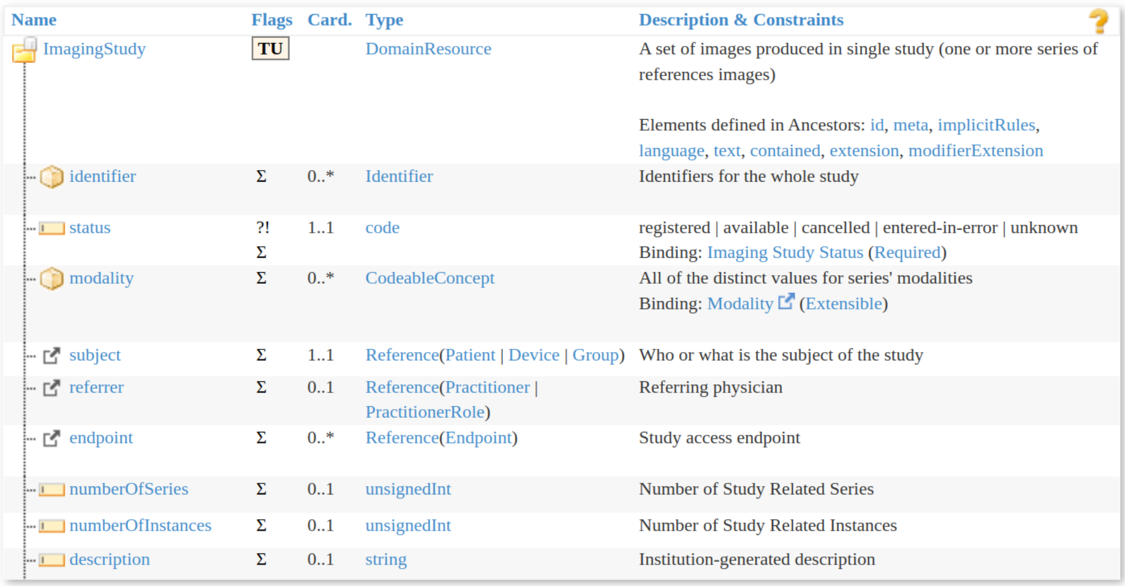
Many technical interoperability standards have emerged over the years. The international FHIR standard (*Fast Healthcare Interoperability Resources*) is nowadays attracting a lot of attention. FHIR is an open, modern standard for exchanging clinical information electronically (Bender and Sartipi, 2013). FHIR is actively developed by the HL7 International organization. The original proposal for FHIR was released in 2011, its first draft standard (referred to as DSTU1, which stands for *First Draft Standard for Trial Use*) was published in 2014, and its first normative content (called Release 4) appeared in 2019.

FHIR is based on a set of standardized resources, which represent discrete pieces of healthcare data, such as patients, medications, allergies, and observations. These resources act as building blocks that can be combined to represent complex clinical concepts and workflows. Furthermore, FHIR employs REST APIs (*Representational State Transfer Application Programming Interfaces*) to access and manipulate the various resources. The fact that REST APIs are widely used for the development of Web and mobile applications explains the popularity of FHIR among software engineers. FHIR also fully embraces existing medical terminologies such as SNOMED-CT and LOINC, which enables semantic interoperability, ensuring that healthcare data is understood consistently across different systems and organizations.

On the other hand, in the context of medical imaging, DICOM (*Digital Imaging and Communications in Medicine*) has been the *de facto* international standard for the encoding, transmission, storage, and sharing of digital images since 1985 (NEMA, 2023). In the DICOM workflow, imaging modalities create DICOM instances that are centralized in so-called PACS servers (*Picture Archiving and Communication System*), which in turn provide features for the viewing and distribution of the collected medical images. DICOM is adopted by any hospital in the world, and its

[a] https://orcid.org/0000-0001-6685-7398

Jodogne, S.
Setting a PACS on FHIR.
DOI: 10.5220/0012384600003657
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 17th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2024) - Volume 2*, pages 123-131
ISBN: 978-989-758-688-0; ISSN: 2184-4305
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.



| Name | Flags | Card. | Type | Description & Constraints |
|---|---|---|---|---|
| ImagingStudy | TU | | DomainResource | A set of images produced in single study (one or more series of references images)<br><br>Elements defined in Ancestors: id, meta, implicitRules, language, text, contained, extension, modifierExtension |
| identifier | Σ | 0..* | Identifier | Identifiers for the whole study |
| status | ?! Σ | 1..1 | code | registered \| available \| cancelled \| entered-in-error \| unknown<br>Binding: Imaging Study Status (Required) |
| modality | Σ | 0..* | CodeableConcept | All of the distinct values for series' modalities<br>Binding: Modality (Extensible) |
| subject | Σ | 1..1 | Reference(Patient \| Device \| Group) | Who or what is the subject of the study |
| referrer | Σ | 0..1 | Reference(Practitioner \| PractitionerRole) | Referring physician |
| endpoint | Σ | 0..* | Reference(Endpoint) | Study access endpoint |
| numberOfSeries | Σ | 0..1 | unsignedInt | Number of Study Related Series |
| numberOfInstances | Σ | 0..1 | unsignedInt | Number of Study Related Instances |
| description | Σ | 0..1 | string | Institution-generated description |

Figure 1: Excerpt of the main structure of the `ImagingStudy` resource in FHIR 5.0.0. For each field in the resource, the "Card." column indicates its cardinality, and the "Type" column indicates the type of its values. Fields whose minimum cardinality is 1 are mandatory. This is a screenshot of the official specification of FHIR available at: https://hl7.org/fhir/R5/ImagingStudy.html.

recent DICOMweb extensions have progressively introduced REST APIs for medical imaging since the 2010s.

Evidently, because of the universal adoption of DICOM by hospitals, researchers, and vendors, FHIR does not try to introduce a new competing standard dedicated to medical imaging. Instead, FHIR defines the `ImagingStudy` resource as its official way to create a connection between clinical and imaging data. This resource allows FHIR systems to reference DICOM entities stored in separate PACS servers through the DICOMweb API. FHIR-to-DICOM references are used to link patients to medical images. Intuitively, the `ImagingStudy` resource can be thought of as a higher-level service on the top of DICOMweb.

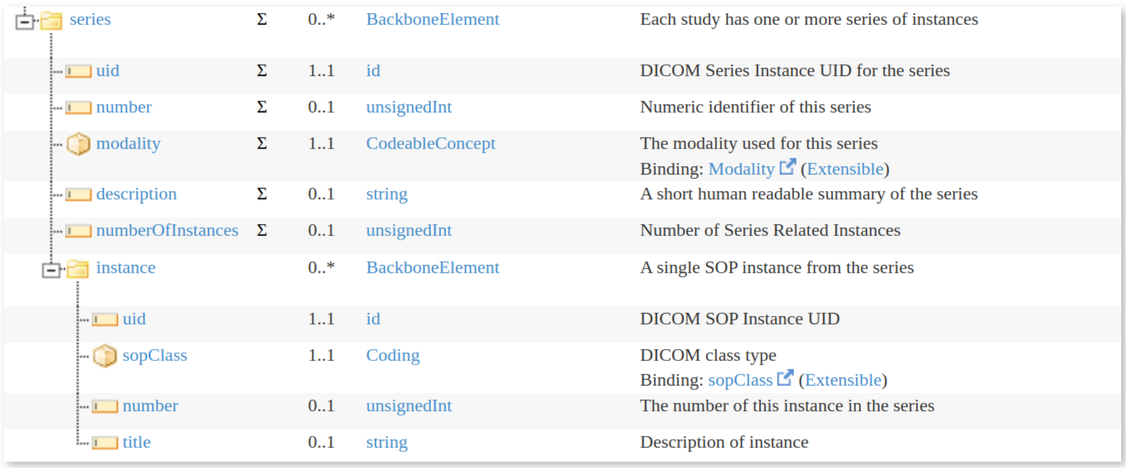
Unfortunately, there is currently a lack of open reference software implementation combining FHIR with DICOM. Even though implementations of the `ImagingStudy` resource do exist and are available in public test sandboxes for FHIR, their content is managed separately from the content of a PACS server. The availability of a DICOM server implementing both the DICOMweb and the FHIR REST APIs could hopefully promote the support of medical imaging in FHIR, while providing a testing environment for the development of new applications combining clinical with imaging data. The difficulty in creating such an implementation lies in the fact that both FHIR and DICOM are rich standards whose implementations are usually developed by separate teams. Furthermore, it is needed to create an infrastructure that continuously keeps FHIR resources in synchronization with DICOM instances. To fill this gap, in this paper, a free and open-source PACS server supporting DICOMweb is extended to publish the parts of the FHIR REST API that are related to medical imaging, as a complement to the DICOM primitives that are already offered by this PACS server.

## 2 BACKGROUND

This section reviews the different building blocks that will be combined to create an open reference implementation of a FHIR server for medical imaging.

### 2.1 DICOM Model of the Real World

In the DICOM model of the real world, a patient benefits from a sequence of imaging studies during his or her lifetime. Each of these studies consists of a set of series. In turn, each series is made of a set of DICOM instances that are created by the same imaging device. An isolated DICOM instance can often (but not always) be thought of as one 2D image slice that is associated with a dataset providing patient-related and acquisition-related information, which enriches the pixel data. This dataset is a tree-like structure that associates values to normalized DICOM tags, identified by a pair of hexadecimal numbers. For instance, a



| | | | | |
|---|---|---|---|---|
| series | Σ | 0..* | BackboneElement | Each study has one or more series of instances |
| uid | Σ | 1..1 | id | DICOM Series Instance UID for the series |
| number | Σ | 0..1 | unsignedInt | Numeric identifier of this series |
| modality | Σ | 1..1 | CodeableConcept | The modality used for this series<br>Binding: Modality (Extensible) |
| description | Σ | 0..1 | string | A short human readable summary of the series |
| numberOfInstances | Σ | 0..1 | unsignedInt | Number of Series Related Instances |
| instance | | 0..* | BackboneElement | A single SOP instance from the series |
| uid | | 1..1 | id | DICOM SOP Instance UID |
| sopClass | | 1..1 | Coding | DICOM class type<br>Binding: sopClass (Extensible) |
| number | | 0..1 | unsignedInt | The number of this instance in the series |
| title | | 0..1 | string | Description of instance |

Figure 2: Excerpt of the information to be provided in the `ImagingStudy` for each of its associated DICOM series. This screenshot is the continuation of Figure 1 in the FHIR standard.

DICOM study generated by a PET-CT-scanner would typically correspond to a set of two DICOM series (representing the PET volume and the CT volume), with the two series encoded as a set of DICOM instances that contain the individual 2D axial slices of the parent 3D volume.

Each resource in the patient/study/series/instance hierarchy can be identified by one specific DICOM tag. The patient level is identified by the value of the `Patient ID` (0x0010,0x0020) DICOM tag, the study level by the `Study Instance UID` (0x0020,0x000d) tag, the series level by the `Series Instance UID` (0x0020,0x000e) tag, and finally the instance level by the `SOP Instance UID` (0x0008,0x0018) tag. PACS systems typically index the DICOM instances they store according to the value of these four important DICOM tags.

## 2.2 FHIR for Medical Imaging

As explained in the Introduction, FHIR brings support for medical imaging through its `ImagingStudy` resource. `ImagingStudy` is associated with the maturity level 4 in Release 5.0.0 of FHIR[1]. This maturity level indicates that `ImagingStudy` has been implemented in multiple prototype projects, hereby showing a certain level of stability. Nonetheless, this also means that the resource has been implemented in less than five independent production systems, or in no more than one country yet. This fact reinforces the interest in developing an open implementation of `ImagingStudy` to raise the maturity level of the associated part of the FHIR standard.

The main content of the `ImagingStudy` resource is depicted in Figure 1. One instance of this FHIR resource corresponds to one DICOM study. To create the link between the `ImagingStudy` resource and its associated DICOM study, the `identifier` field of the FHIR resource must contain the value of the `Study Instance UID` DICOM tag. Very importantly, this means that FHIR does not store the actual images, but rather pointers to DICOM resources. The resources themselves must be stored in a separate DICOMweb-compliant server, the address of which must be provided in the `endpoint` field. The latter field consists of a reference to a separate `Endpoint` FHIR resource that specifies the connection parameters to the DICOMweb server, including its URL. Existing sandboxes for FHIR typically do not contain this `endpoint` field, which prevents access to the raw DICOM studies that are indexed by the `ImagingStudy` resources.

As can be seen in Figure 1, besides the `Endpoint` and `ImagingStudy` resources, a FHIR server for medical imaging must also support the `Patient` resource, because the `subject` field must contain a reference to the patient who is associated with the DICOM study. Consequently, similarly to the one-to-one relation that exists between a FHIR `ImagingStudy` and a DICOM study, there exists a mapping between a `Patient` FHIR resource and a DICOM patient.

In addition to the patients and to the studies, FHIR for medical imaging also creates links with both the DICOM series and the DICOM instances. To this end, the `ImagingStudy` resource contains a field named `series`, as shown in Figure 2. This field stores an array, each item of which declares one DICOM series whose `Series Instance UID` DICOM tag is con-

[1]FHIR 5.0.0 is the most recent version of FHIR at the time of writing. It corresponds to the version of FHIR that is used throughout this work.

tained in the `uid` subfield. In turn, each item in the `series` field contains a subarray called `instance` that provides the list of the DICOM instances of the parent series, in which the `uid` subfield indicates the `SOP Instance UID` DICOM tag of the individual instances. This means that, contrary to DICOM studies that are mapped as separate `ImagingStudy` resources, FHIR does not encode DICOM series and DICOM instances as standalone resources: The series/instance information can only be retrieved by accessing their parent study.

Previous work about the `ImagingStudy` resource has consisted in providing static databases as pedagogical resources to learn FHIR for medical imaging (Hussain et al., 2018), in deploying a multi-site infrastructure dedicated to pediatric scoliosis (Shi et al., 2021), in collecting radiation dose information (Boufahja et al., 2021), and in implementing complex clinical imaging workflows (Tang et al., 2023). However, to the best of our knowledge, no free and open-source PACS server can currently act as a standalone FHIR server that transparently, automatically publishes its DICOM resources using DICOMweb.

### 2.3 HAPI Framework

The most complete implementation of the FHIR standard is HAPI (Ayaz et al., 2021). HAPI is free and open-source software that is licensed under the Apache Software License 2.0 and that is written in the Java programming language[2]. HAPI closely follows the release cycle of the FHIR standard. It can be used as a building block to design both FHIR clients and FHIR servers. Importantly, HAPI comes with support for the `ImagingStudy` resource that is needed to implement medical imaging in FHIR.

The HAPI framework proposes two approaches for designing FHIR servers. On the one hand, the "HAPI JPA Server" is a fully standalone application that incorporates the Java Persistence API (JPA) to store and retrieve FHIR resources in a relational database such as PostgreSQL. On the other hand, the HAPI project also provides the so-called "HAPI Plain Server" as a software library that can be used to create custom FHIR endpoints against arbitrary data sources. This paper will focus on the combination of the HAPI framework with a PACS server to provide a tight integration between FHIR and DICOM.

[2]The homepage of the HAPI project is available at: http://hapifhir.io/.

### 2.4 Orthanc Server

In the context of medical imaging, multiple free and open-source DICOM servers are available, such as dcm4chee (Warnock et al., 2007) and Dicoogle (Costa et al., 2010). This work is focused on the Orthanc DICOM server, which is licensed under the GPLv3 license and is developed in the C++ language (Jodogne, 2018). Besides its DICOM primitives, Orthanc comes with a REST API that can be used to script imaging workflows, which is widely used to set up auto-routing between local DICOM servers or between remote sites. This REST API can notably be used to retrieve the content of individual DICOM resources and to search the content of the DICOM database, which will be used in this work. Orthanc has the advantage of being lightweight and extensible: Multiple Orthanc servers can run on one single computer at low cost, without necessitating any complex configuration, and the core features of Orthanc can be extended through plugins that take the form of shared libraries.

Importantly, the fact that Orthanc plugins can extend the REST API of Orthanc has already been used to integrate Web viewers such as OHIF (Ziegler et al., 2020) and VolView (Xu et al., 2022), as well as to provide an implementation of the DICOMweb standard. In this work, the extensibility of Orthanc will be exploited to seamlessly combine it with HAPI, while providing an architecture that is as easy to deploy as possible on any computer by leveraging the small footprint of Orthanc.

## 3 METHODS

In this section, a software architecture that integrates Orthanc with HAPI is designed. This architecture is based on the plugin system of Orthanc.

### 3.1 High-Level Architecture

Figure 3 depicts three possible high-level architectures to combine HAPI with Orthanc. The first architecture consists in executing the HAPI JPA Server next to the Orthanc server, in two separate processes. In this architecture, a third process is used to ensure the continuous synchronization between HAPI and Orthanc. To this end, the synchronization process monitors the addition and removal of DICOM resources using the REST API of Orthanc, and replicates the detected modifications in the HAPI JPA Server. In this architecture, the FHIR and DICOM databases are kept separate, which results in a substantial risk of de-synchronization between the two

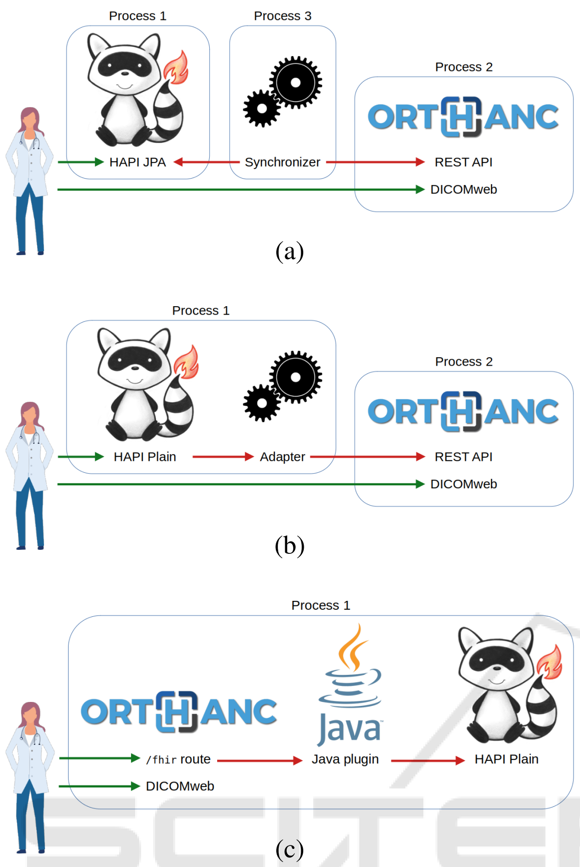


Figure 3: Three possible architectures to integrate the Orthanc server with the HAPI framework: (a) continuous synchronization between Orthanc and HAPI JPA Server, (b) custom HAPI Plain Server with a data source corresponding to Orthanc, or (c) branching HAPI Plain Server as a plugin to Orthanc.

servers, and which requires a specific infrastructure dedicated to the execution of HAPI JPA Server.

The second possible architecture consists in replacing HAPI JPA Server by HAPI Plain Server. In this architecture, a custom HAPI Plain Server is deployed in a standalone Java servlet container and acts as a facade in front of the Orthanc server. This facade is responsible for generating the `ImagingStudy` and `Patient` FHIR resources after querying the REST API of Orthanc. This solution has the great advantage of using a single database that is shared by the FHIR and DICOM servers. However, this architecture implies the deployment of two distinct processes and two distinct Web servers (one for the servlet container, and one for Orthanc), which keeps the setup slightly complex, especially when it comes to the mapping of the DICOMweb API.

Therefore, this work is focused on a third architecture, in which HAPI Plain Server is directly branched behind the Web server of Orthanc. In this architecture, Orthanc acts similarly to a servlet container, by redirecting all the FHIR requests to the HAPI framework. This architecture has the advantage of necessitating not only one single database, but also one single process (which facilitates the setup for newcomers), as well as of providing uniform access to FHIR resources and to DICOMweb routes.

## 3.2 Java Plugins

The difficulty with the third architecture that was selected in Section 3.1 is that the Orthanc server is developed in C++, which necessitates Orthanc plugins to be written in C or C++. This contrasts with the HAPI framework that is written in Java. To provide compatibility between Orthanc and HAPI, a framework to create Orthanc plugins in Java has been designed by leveraging the JNI (*Java Native Interface*).

More precisely, a native C++ plugin for Orthanc has been created that starts a Java virtual machine from within Orthanc using JNI. This native plugin first loads the main Java class that is specified in the configuration file of Orthanc, which gives the opportunity to this main Java class to register Java handlers in order to react to the various events that are processed by Orthanc. In particular, the main Java class can install handlers to answer HTTP requests that are received by the Web server of Orthanc. The native C++ plugin maintains the list of the registered Java handlers and invokes these handlers once the corresponding callback is triggered by Orthanc.

Besides the support of callbacks to react to events, the Orthanc plugin SDK (*Software Development Kit*) proposes a large library of C functions that can be used to invoke the primitives implemented by the Orthanc core. In particular, the Orthanc plugin SDK gives the possibility to make internal calls to the REST API of Orthanc, which can be used to achieve a wide variety of operations. The Java handlers need to use this library of C functions.

To this end, two Python scripts have been implemented to automatically wrap the Orthanc plugin SDK as a collection of Java classes and methods. The first Python script first analyzes the content of the `OrthancCPlugin.h` C header file that defines the Orthanc plugin SDK, then creates a code model that is saved in the JSON file format. This script automatically discovers the classes that are managed by the Orthanc core and that are accessible through the SDK (such as DICOM instances, images, or responses to REST requests). The resulting code model also includes the documentation. This analysis is achieved using the `libclang` library that is part of the LLVM project (Lattner and Adve, 2004).

Once the code model is extracted, a second Python script generates the Java classes that act as wrappers around the native C objects (including the calls to their destructors), as well as a specific Java class named `Functions` that contains the wrappers around the global C functions that do not correspond to objects. A Javadoc-compliant documentation is generated as well. This second script also generates a C++ file whose goal consists in associating the Java native methods with their concrete implementation provided by the Orthanc plugin SDK through JNI.

Thanks to this automated mechanism, 122 functions from the Orthanc plugin SDK are automatically wrapped in Java out of a total of 160 primitives available in Orthanc 1.10.0, without any human intervention, which would have been highly time-consuming and error prone. This represents a coverage of about 76%. The primitives that are not automatically wrapped mostly correspond to the handling of callbacks, which must be manually implemented. Note that the extracted code model could be used to wrap the Orthanc plugin SDK in languages other than Java.

### 3.3 Integrating HAPI

The native plugin presented in Section 3.2 enables the use of the HAPI framework inside Orthanc. Nonetheless, HAPI Plain Server is based on Java servlets that are not directly compatible with the primitives offered by the Orthanc plugin SDK. The solution to this issue consists in using the `MockHttpServletRequest` and `MockHttpServletResponse` classes that are provided by the Spring Framework. These two standard classes can be used to turn the Web server of Orthanc into a servlet container: Whenever a HTTP request must be handled by a Java plugin, an instance of the `MockHttpServletRequest` class is filled with the parameters of the HTTP request and is sent to the Java servlet. In turn, the Java servlet fills a `MockHttpServletResponse` whose content can be unwrapped and sent to the client through the Orthanc Web server. In the context of the FHIR plugin for Orthanc, the HAPI Plain Server servlet is by default branched at the root path `/fhir/` in the REST API of Orthanc.

The last step to integrate HAPI within Orthanc consists in implementing read-only providers that are responsible to retrieve and search the `ImagingStudy`, `Patient`, and `Endpoint` FHIR resources. To this end, the FHIR plugin for Orthanc implements the `IResourceProvider` interface provided by HAPI for each of those resources. The `Endpoint` resource simply declares the address of the DICOMweb plugin, which is a purely static behavior that does not require querying the content of the Orthanc database.

As far as the `ImagingStudy` (resp. `Patient`) resources are concerned, the content of an individual DICOM study (resp. DICOM patient) with identifier `id` can be retrieved by GET-ing the `/studies/{id}` (resp. `/patients/{id}`) route of the built-in REST API of Orthanc. This route returns a JSON object whose content can easily be converted as an `ImagingStudy` (resp. `Patient`) Java object that is expected by the HAPI framework.

To implement searching over the `ImagingStudy` and `Patient` FHIR resources, the FHIR plugin for Orthanc issues POST requests to the `/tools/find` route of the built-in REST API of Orthanc. This route implements a lookup in the Orthanc database, which can be constrained by specifying values for selected DICOM tags. These constraints can directly be derived from the parameters of the FHIR query. The route answers with a JSON array that contains the matching resources, which can easily be converted into the `List<ImagingStudy>` and `List<Patient>` Java objects that are expected by the HAPI framework.

## 4 RESULTS

Section 3 explained how HAPI can be started as a plugin to Orthanc, which provides a working implementation of a FHIR server for medical imaging that is tightly integrated with a PACS. As a byproduct of this research, Orthanc plugins can now be developed in the Java programming language. An example of a Java plugin and an interaction with the FHIR server plugin for Orthanc are now discussed.

### 4.1 Using dcm4che in Orthanc

The dcm4chee server was previously mentioned as a free and open-source implementation of a PACS. The dcm4chee server is built on the top of dcm4che, a highly popular software library that provides an advanced implementation of the DICOM standard in Java. This contrasts with Orthanc that uses the DCMTK library written in C++. Consequently, the new Java plugin for Orthanc can be used to complement DCMTK with dcm4che, which allows to benefit from the respective strengths of these two software libraries inside the Orthanc ecosystem. Depending on his or her background, a software developer might also find it easier to develop a plugin in Java than in C++.

As an illustration, Figure 4 shows a sample Java

```java
import org.dcm4che3.data.Attributes;
import org.dcm4che3.io.DicomInputStream;

public class Main {
    static {
        Callbacks.register("/dcm4che-parse", new Callbacks.OnRestRequest() {
            @Override
            public void call(RestOutput output,
                             HttpMethod method,
                             String uri,
                             String[] regularExpressionGroups,
                             Map<String, String> headers,
                             Map<String, String> getParameters,
                             byte[] body) {
                if (method != HttpMethod.POST) {
                    output.sendMethodNotAllowed("POST");  // Answer with HTTP status 405
                } else {
                    ByteArrayInputStream stream = new ByteArrayInputStream(body);

                    try (DicomInputStream din = new DicomInputStream(stream)) {
                        Attributes dataset = din.readDataset();
                        output.answerBuffer(dataset.toString().getBytes(), "text/plain");
                    } catch (IOException e) {
                        output.sendHttpStatus((short) 400, "Cannot parse DICOM file\n".getBytes());
                    }
                }
            }
        });
    }
}
```

Figure 4: Example of a Java plugin for Orthanc that uses dcm4che.

plugin that uses the dcm4che library. The configuration file of Orthanc must specify both the classpath for the Java virtual machine (which must contain the `.jar` files providing dcm4che and the bytecode of `Main.class`), and the name of the main Java class whose `static` method will be executed during the initialization of the plugin. In this sample `Main` plugin, the `static` method registers a new route in the REST API of Orthanc using the static method `Callbacks.register()` that is implemented by the Java SDK of Orthanc. Whenever a HTTP request is received by Orthanc against the `/dcm4che-parse` route in its REST API, the Java plugin is invoked. Inside the Java plugin, the body of incoming HTTP POST requests is parsed as a DICOM instance using dcm4che. On success, the client receives the dump of the DICOM dataset, as generated by dcm4che. In a sense, such a Java plugin can be considered as a servlet that extends the REST API of Orthanc.

## 4.2 FHIR Server in Orthanc

Starting the FHIR server plugin for Orthanc simply consists in writing a configuration file that loads the `.jar` containing both the HAPI framework and the Java plugin implementing the FHIR server. A single precompiled `.jar` file that packages all the required dependencies is freely available for download on the Orthanc official homepage. This package is generated by Maven and its `maven-assembly-plugin` plugin. As soon as Orthanc is running, its FHIR server is accessible at default URL: http://localhost:8042/fhir/. The FHIR clients can access the content of Orthanc using this URL. The `Endpoint` resource of the FHIR server is automatically mapped to the route of the DICOMweb server of Orthanc, whose default location corresponds to: http://localhost:8042/dicom-web/studies. This means that only one single Web server is running at any time, with this Web server being shared by the REST API of Orthanc, by the DICOMweb endpoint, and by HAPI Plain Server.

As an illustration, Figure 5 shows an interactive session between the FHIRPACK client[3] and the FHIR server plugin for Orthanc. Importantly, the reported FHIR resources (`Patient` and `ImagingStudy`) are generated on-the-fly using HAPI Plain Server, directly from the DICOM resources that are stored by Orthanc: No separate database dedicated to FHIR

[3]The homepage of the FHIRPACK project is available at: https://github.com/fhirpack/fhirpack

```
$ fp -s http://localhost:8042/fhir -o "getPatients" -p all -o "gatherSimplePaths id name.family birthDate"

0      fYET5.0   [COMUNIX]  1941-09-01
1        5Yp0E   [BRAINIX]  1949-03-01
2     Vafk,T,6    [PHENIX]  1991-01-01
3       SOtNwu   [INCISIX]        None
4  ozp00SjY2xG      [KNIX]        None
5       vAD7q3       [VIX]        None


$ fp -s http://localhost:8042/fhir -o "getImagingStudies" -p all -o \
  "gatherSimplePaths identifier.value subject.reference description numberOfSeries numberOfInstances"

                                identifier.value    subject.reference                 description  numberOfSeries  numberOfInstances
0  [urn:oid:2.16.840.1.113669.632.20.1211.1000031...       Patient/vAD7q3  Pied_cheville_UHR (Adulte)               1                250
1  [urn:oid:2.16.840.1.113669.632.20.1211.1000035...        Patient/5Yp0E  IRM cérébrale, neuro-crâne               7                232
2  [urn:oid:2.16.840.1.113669.632.20.1211.1000009...     Patient/Vafk,T,6       CT2 tête, face, sinus               3                723
3  [urn:oid:1.2.840.113745.101000.1008000.38048.4...      Patient/fYET5.0       Neck^1HEAD_NECK_PETCT               2                166
4  [urn:oid:1.2.840.113619.2.176.2025.1499492.739...  Patient/ozp00SjY2xG                    Knee (R)               6                135
5  [urn:oid:2.16.840.1.113669.632.20.1211.1000023...       Patient/SOtNwu        Tête^Dental (Adulte)               1                166
```

Figure 5: Sample interactive session of the FHIRPACK client against an Orthanc server that contains sample DICOM studies provided by the OsiriX project (Rosset et al., 2004). The first request lists the `Patient` FHIR resources, while the second lists the `ImagingStudy` FHIR resources.

resources is used, all the information is extracted from the Orthanc database. This ensures that the DICOMweb and the FHIR views of the content of the Orthanc server are always perfectly synchronized.

### 4.3 Software Availability

These new contributions are released as free and open-source software under the GPLv3 license, which corresponds to the license of the Orthanc server. The software associated with this research contains more than 14,000 lines of code, as reported by the `cloc` command-line tool, which is an indication of its complexity. The source code of both the Java plugin and the FHIR server for Orthanc is available at: https://orthanc.uclouvain.be/hg/orthanc-java/. The official download site of Orthanc provides precompiled binaries and installers for the Java plugin and for the FHIR server. Full technical documentation is available inside the Orthanc Book, the official documentation of the Orthanc project.

## 5 CONCLUSIONS

This paper proposes a free and open-source framework to run a FHIR server for medical imaging alongside a DICOM server. This is achieved by combining the Orthanc server with HAPI Plain Server through a Java plugin for Orthanc. Both the FHIR resources and the DICOMweb API are served through the Web server of Orthanc, using one single process, which simplifies the deployment and provides a robust environment. Furthermore, the internal architecture of the solution guarantees that FHIR and DICOM resources are synchronized at any time. Thanks to the lightness of Orthanc, the application can easily be executed on any computer equipped with a Java virtual machine. As a byproduct, it is also shown how Java can be used to create plugins that extend the core features of Orthanc.

The resulting software is hopefully a suitable candidate to promote the `ImagingStudy` FHIR resource and hence to raise its maturity level. Even though `ImagingStudy` is already supported by other open-source implementations of FHIR, most notably HAPI, these implementations are not tightly coupled with the content of a DICOMweb server. Consequently, our contributions could serve as a test bed for software engineers to develop new, innovative, and interoperable solutions for medical imaging. It could also be used as a pedagogical platform to teach the principles of modern health interoperability. The main limitation of the designed software is that it currently only implements the `ImagingStudy`, `Patient`, and `Endpoint` resources: It would especially be interesting to create a bridge between FHIR and DICOM worklists using the `Task` and `ServiceRequest` resources. Future work will also expand the FHIR support in Orthanc with a focus on artificial intelligence and will take advantage of FHIR in the context of medical research combining clinical, imaging, and multi-omics data.

## REFERENCES

Ayaz, M., Pasha, M. F., Alzahrani, M. Y., Budiarto, R., and Stiawan, D. (2021). The fast health interoperability resources (FHIR) standard: Systematic literature review of implementations, applications, challenges and opportunities. *JMIR Medical Informatics*, 9(7):e21929.

Bender, D. and Sartipi, K. (2013). HL7 FHIR: An agile and RESTful approach to healthcare information exchange. In *Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems*. IEEE.

Benson, T. and Grieve, G. (2021). *Principles of Health Interoperability*. Springer International Publishing.

Boufahja, A., Nichols, S., and Pangon, V. (2021). Custom FHIR resources definition of detailed radiation information for dose management systems. In Pesquita, C., Fred, A. L. N., and Gamboa, H., editors, *Proceedings of the 14th International Joint Conference on Biomedical Engineering Systems and Technologies, BIOSTEC 2021, Volume 5: HEALTHINF, Online Streaming, February 11-13, 2021*, pages 467–474. SCITEPRESS.

Costa, C., Ferreira, C., Bastião, L., Ribeiro, L., Silva, A., and Oliveira, J. L. (2010). Dicoogle: An open-source peer-to-peer PACS. *Journal of Digital Imaging*, 24(5):848–856.

Hussain, M. A., Langer, S. G., and Kohli, M. (2018). Learning HL7 FHIR using the HAPI FHIR server and its use in medical imaging with the SIIM dataset. *Journal of Digital Imaging*, 31(3):334–340.

Jodogne, S. (2018). The Orthanc ecosystem for medical imaging. *Journal of Digital Imaging*, 31(3):341–352.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California.

NEMA (2023). National Electrical Manufacturers Association PS3 / ISO 12052, Digital Imaging and Communications in Medicine (DICOM) standard. http://www.dicomstandard.org/.

Rosset, A., Spadola, L., and Ratib, O. (2004). OsiriX: An open-source software for navigating in multidimensional DICOM images. *Journal of Digital Imaging*, 17(3):205–216.

Shi, W., Giuste, F. O., Zhu, Y., Carpenter, A. M., Iwinski, H. J., Hilton, C., Wattenbarger, J. M., and Wang, M. D. (2021). A fhir-compliant application for multi-site and multi-modality pediatric scoliosis patient rehabilitation. In Huang, Y., Kurgan, L. A., Luo, F., Hu, X., Chen, Y., Dougherty, E. R., Kloczkowski, A., and Li, Y., editors, *IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2021, Houston, TX, USA, December 9-12, 2021*, pages 1524–1527. IEEE.

Tang, S.-T., Tjia, V., Noga, T., Febri, J., Lien, C.-Y., Chu, W.-C., Chen, C.-Y., and Hsiao, C.-H. (2023). Creating a medical imaging workflow based on FHIR, DICOMweb, and SVG. *Journal of Digital Imaging*, 36(3):794–803.

Warnock, M., Toland, C., Evans, D., Wallace, B., and Nagy, P. (2007). Benefits of using the DCM4CHE DICOM archive. *Journal of Digital Imaging*, 20(S1):125–129.

Xu, J., Thevenon, G., Chabat, T., McCormick, M., Li, F., Birdsong, T., Martin, K., Lee, Y., and Aylward, S. (2022). Interactive, in-browser cinematic volume rendering of medical images. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization*, 11(4):1019–1026.

Ziegler, E., Urban, T., Brown, D., Petts, J., Pieper, S. D., Lewis, R., Hafey, C., and Harris, G. J. (2020). Open health imaging foundation viewer: An extensible open-source framework for building web-based imaging applications to support cancer research. *JCO Clinical Cancer Informatics*, (4):336–345.