ISR & INESC-ID

TECHNICAL MANUAL (EN)

MIMBCD-UI

Contents

Intro

This Technical Manual is intended to be used as a reference for the concepts and API's in MIMBCD-UI systems, more specifically, the *prototype-cornerstone* repository. If you are new to the system, we recommend that you start by reviewing the concepts below and then looking at the various examples to see how cornerstone is used. It is also useful to read the Master Thesis of Francisco Maria Calisto[1] or a related work[2].

[1]

[2]

System Concepts

- **Architecture**
- **Image Processing**
- **Viewport**

Architecture

Some indiscriminate thoughts on the architecture where advantages on other libraries that where possible to minimize the amount of code to debug and maintain. In fact, no web framework shall be utilized such as AngularJS, Backbone, EmberJS or even Meteor. The *prototype-cornerstone* is focused on a layer lower than these frameworks and should not need them. Also, *prototype-cornerstone* should be useable with these frameworks though and possibly include adapters that make it easier to use in those frameworks.

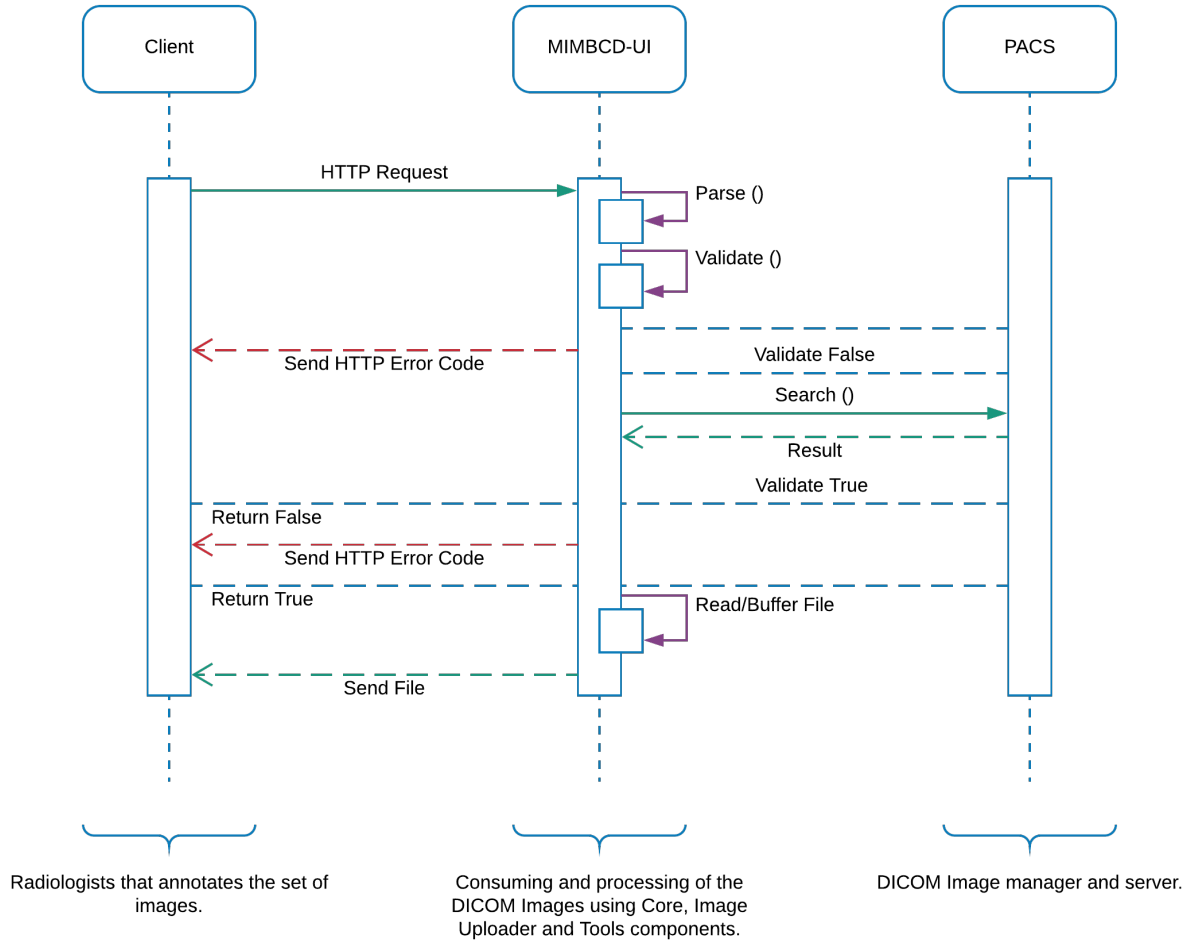
Do not use libraries that force dependencies on users of the library. For example, it should be possible for an app that uses a different version of a library than cornerstone to work properly. Prefer small simple libraries to bigger ones. For instance, jQuery is too big. It should be possible to share the same library (as long as the version is supported by both).

Asynchronous APIs should leverage the deferred pattern to return promises that callers can use to properly handle success and failure conditions. The excellent Cujo.js when library has great documentation about this (and is used by our engine along with other cujo.js libraries such as REST and CURL). The API should present a simplified view of the medical imaging domain to make it easier for developers to work with.

The *prototype-cornerstone* can be framed into the diagram shown in the next image. More specifically, the implementation of the (i) **Client** service; accessed by the (ii) **MIMBCD-UI** services; and served by a (iii) **PACS** server (*Orthanc*). Part of the DICOM standard must be accepted as a service of the **PACS** server type and must be the response answer of the **MIMBCD-UI** services, in order to determine the parameters. Additionally, the standard includes an exception with respect to several parameters.

We will first describe the (iii) **PACS** server (*Orthanc*), since we first upload the DICOM images through this system. Second, we will describe the (ii) **MIMBCD-UI** services, while it uses several libraries. Third, we will describe the (i) **Client**. The standard defines parameters in two variants; required and optional. If a **PACS** server (*Orthanc*) supports search by unique instance identifier, parameters of study and series are not necessary. The required parameters are those that identify unequivocally a DICOM object; the study identifiers, series and instance. All other parameters refer to modifications on the image or format of the data. This is the case of a server, which our system is implemented in, so the amount of parameters that should be treated and validated when doing searches, are reduced. The next image shows an UML sequence corresponding to with the execution of an **HTTP Request** on the **MIMBCD-UI** (WADO) services. The design of the service is explained below. Among these tools we can find support of Cornerstone [3] libraries. These libraries have in common being based in HTML5 and JavaScript so they allow the manipulation, display and access to DICOM object information completely in a web browser (**Client**).

[3]



The ways of clarification, for the integration capacity are considered mainly two factors, if the tool allow us to easily configure it for an existing **PACS** and if the application type is compatible with an AWS server like an EC2. This last aspect is only considered for the convenience that the **PACS** services used with **MIMBCD-UI** System services are deployed on an AWS server. As a research of this study it was decided to explore the Cornerstone libraries to create a web-based DICOM viewer. While it is a solution that requires greater implementation effort than those studied, which offer a fully functional image viewer, it is a basis for with a multimodality of medical imaging capabilities. Cornerstone is a JavaScript library designed to show medical images in a web environment by doing use of the HTML5 Canvas element. Cornerstone Core is not meant to be a complete application itself, but instead a component that can be used as part of larger more complex applications. The **MIMBCD-UI** System services are an example of using the various Cornerstone libraries to build a simple study viewer.

Cornerstone Core is agnostic to the actual container used to store image pixels and to transport mechanisms used to get the image data. As a matter of fact, Cornerstone Core itself has no ability to read/-parse or load images and instead depends on one or more *ImageLoader* and *dicomParser* (**Validate ()** and **Parse ()** from the last figure to function. The goal here is to avoid constraining developers to work within a single container and transport since images are stored in a variety of formats (e.g. DICOM). By providing flexibility with respect to the container and transport, the highest performance image display may be obtained as no conversion to an alternate container or transport is required. It is hoped that developers feel empowered to load images from any type of image container using any kind of transport. See the *CornerstoneWADOImageLoader* repository for an example of a DICOM WADO based Image Loader.

The *prototype-cornerstone* often have associated data with it. Several tools might need to keep track of the starting and ending pixel coordinates of the length measurement and which ImageId it is associated with. The cornerstone tools library addresses this by providing the following API functions to work with tool specific data.

The *addToolState* API call is used to add tool specific data to the tool data context associated with an enabled element.

```
function addToolState(element, toolType, data)
```

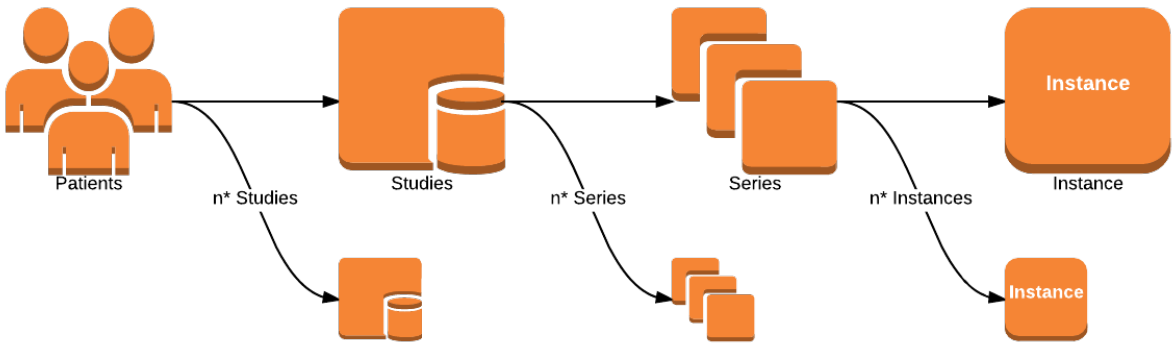
The *getToolState* API call is used to get tool specific data for the tool data context associated with an enabled element.

```
function getToolState(element, toolType)
```

To better understand and to have more details about the hereby information, just follow the *DataManagement* wiki.

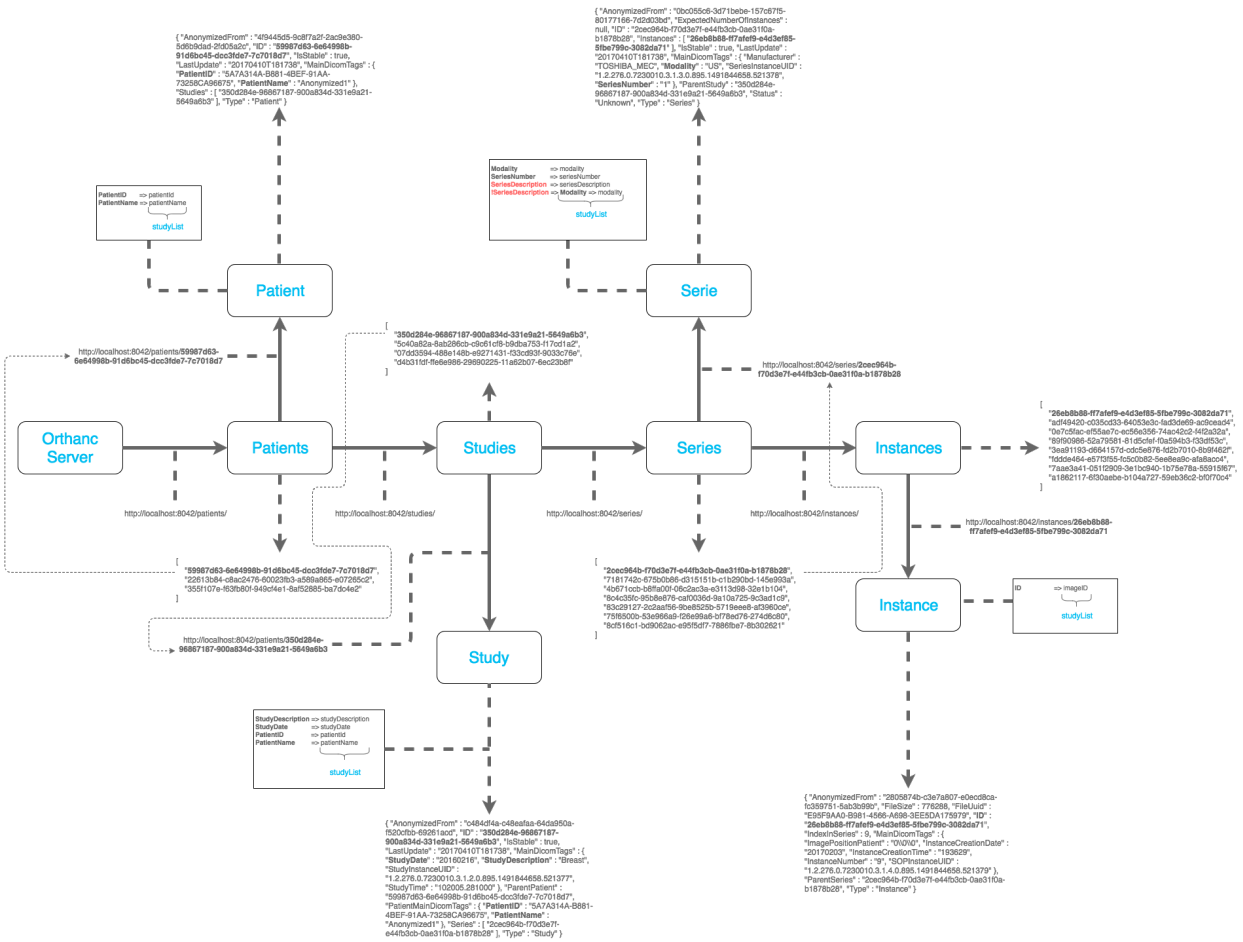
Image Processing

Communication between devices using the DICOM network protocol is done via TCP and IP. To establish a connection, devices or applications negotiate who is the client and who is the server and establish an association. Once associated, orders can be sent to copy, save, delete and move DICOM objects. DICOM objects are not limited to image data, it contains a data structure with a large amount of meta information. Each element within a DICOM object is classified according to a dictionary of identifying tags uniquely a data field. Among these labels, very numerous, there are fields that point to the patient's data (name, date of birth), data of the image and to what study and series the object belongs. To clarify what a series in DICOM is, we must refer to the hierarchy, present in the following diagram. In this standard one patient has multiple studies, one study has multiple series and a series has multiple instances (images, DICOM objects).



One of the main goals of the standard is that DICOM objects can be uniquely identified. On a DICOM objects, the patient's data, the image and other data that relate to one object to the other (such as **Series** and **Studies**) are unequivocally bound. A part of the standard describes how requests and responses should be in a web service that serves DICOM objects and throughout its evolution has incorporated definitions for web services and RESTful API. This part of the standard is of particular interest to the *prototype-cornerstone* since the goal of allowing access to objects like DICOM objects via an UI can be ideally developed by implementing a service that complies with the specifications of the standard.

Our *prototype-cornerstone* has an Image Loader for DICOM P10 instances over HTTP (WADO-URI). Nevertheless, it can also be also be pair with a DICOMWeb (WADO-RS). We are using our *prototype-cornerstone* integrated with WADO-URI servers, a HTTP based server that returns DICOM P10 instances called Orthanc Server. The following Figure shows how we obtained each instance information related to the patients to be read by the **MIMBCD-UI** System on the *prototype-cornerstone* repository.



While Orthanc Servers are therefore built-in RESTful API, it can be used to drive Orthanc from external application, as shown on the above figure. That way Orthanc is independent of the programming language that is used to develop other applications, in our case, for our *prototype-cornerstone*.

To use Orthanc just open the Orthanc Explorer. The embedded administrative interface of Orthanc, with a browser. It entirely resorts to the RESTful API for all its features. So that, anything that can be done through Orthanc Explorer, can also be done trough REST queries.

Patients

The Orthanc Server structures the stored DICOM resources using the **Patient** model of the DICOM standard. Each DICOM resources is associated with an unique identifier:

```
http://localhost:8042/patients/
```

Here is how you would list of **Patients** for the DICOM resources that are stored in your local Orthanc instance:

```
[
   "59987d63-6e64998b-91d6bc45-dcc3fde7-7c7018d7",
   "22613b84-c8ac2476-60023fb3-a589a865-e07265c2",
   "355f107e-f63fb80f-949cf4e1-8af52885-ba7dc4e2"
]
```

To access a single resource, add its identifier to the URI. You would for instance retrieve the main information about one **Patient** as follows:

```
http://localhost:8042/patients/59987d63-6e64998b-91d6bc45-dcc3fde7-7c7018d7
```

Here is a possible answer from Orthanc:

```
{
   "AnonymizedFrom" : "4f9445d5-9c8f7a2f-2ac9e380-5d6b9dad-2fd05a2c",
   "ID" : "59987d63-6e64998b-91d6bc45-dcc3fde7-7c7018d7",
   "IsStable" : true,
   "LastUpdate" : "20170410T181738",
   "MainDicomTags" : {
      "PatientID" : "5A7A314A-B881-4BEF-91AA-73258CA96675",
      "PatientName" : "Anonymized1"
   },
   "Studies" : [ "350d284e-96867187-900a834d-331e9a21-5649a6b3" ],
   "Type" : "Patient"
}
```

Studies

The Orthanc Server structures the stored DICOM resources using the **Study** model of the DICOM standard. Each DICOM resources is associated with an unique identifier:

```
http://localhost:8042/studies/
```

The field **Studies** list all the DICOM studies that are associated with the **Patient**. So, considering the **Patient** above, we would go down in her DICOM hierarchy as follows:

```
[
   "350d284e-96867187-900a834d-331e9a21-5649a6b3",
   "5c40a82a-8ab286cb-c9c61cf8-b9dba753-f17cd1a2",
   "07dd3594-488e148b-e9271431-f33cd93f-9033c76e",
   "d4b31fdf-ffe6e986-29690225-11a62b07-6ec23b8f"
]
```

To access a single resource, add its identifier to the URI. You would for instance retrieve the main information about one **Study** as follows:

```
http://localhost:8042/studies/350d284e-96867187-900a834d-331e9a21-5649a6b3
```

Here is a possible answer from Orthanc:

```
{
   "AnonymizedFrom" : "c484df4a-c48eafaa-64da950a-f520cfbb-69261acd",
   "ID" : "350d284e-96867187-900a834d-331e9a21-5649a6b3",
   "IsStable" : true,
   "LastUpdate" : "20170410T181738",
   "MainDicomTags" : {
      "StudyDate" : "20160216",
      "StudyDescription" : "Breast",
      "StudyInstanceUID" : "1.2.276.0.7230010.3.1.2.0.895.1491844658.521377",
      "StudyTime" : "102005.281000"
   },
   "ParentPatient" : "59987d63-6e64998b-91d6bc45-dcc3fde7-7c7018d7",
   "PatientMainDicomTags" : {
      "PatientID" : "5A7A314A-B881-4BEF-91AA-73258CA96675",
      "PatientName" : "Anonymized1"
   },
   "Series" : [ "2cec964b-f70d3e7f-e44fb3cb-0ae31f0a-b1878b28" ],
   "Type" : "Study"
}
```

Series

The Orthanc Server structures the stored DICOM resources using the **Serie** model of the DICOM standard. Each DICOM resources is associated with an unique identifier:

```
http://localhost:8042/series/
```

The field **Series** list all the DICOM studies that are associated with the **Study**. So, considering the **Study** above, we would go down in her DICOM hierarchy as follows:

```
[
   "2cec964b-f70d3e7f-e44fb3cb-0ae31f0a-b1878b28",
   "7181742c-675b0b86-d315151b-c1b290bd-145e993a",
   "4b671ccb-b8ffa00f-06c2ac3a-e3113d98-32e1b104",
   "8c4c35fc-95b8e876-caf0036d-9a10a725-9c3ad1c9",
   "83c29127-2c2aaf56-9be8525b-5719eee8-af3960ce",
   "75f6500b-53e966a9-f26e99a6-bf78ed76-274d6c80",
   "8cf516c1-bd9062ac-e95f5df7-7886fbe7-8b302621"
]
```

To access a single resource, add its identifier to the URI. You would for instance retrieve the main information about one **Serie** as follows:

```
http://localhost:8042/series/2cec964b-f70d3e7f-e44fb3cb-0ae31f0a-b1878b28
```

Here is a possible answer from Orthanc:

```
{
   "AnonymizedFrom" : "0bc055c6-3d71bebe-157c67f5-80177166-7d2d03bd",
   "ExpectedNumberOfInstances" : null,
   "ID" : "2cec964b-f70d3e7f-e44fb3cb-0ae31f0a-b1878b28",
   "Instances" : [ "26eb8b88-ff7afef9-e4d3ef85-5fbe799c-3082da71" ],
   "IsStable" : true,
   "LastUpdate" : "20170410T181738",
   "MainDicomTags" : {
      "Manufacturer" : "TOSHIBA_MEC",
      "Modality" : "US",
      "SeriesInstanceUID" : "1.2.276.0.7230010.3.1.3.0.895.1491844658.521378",
      "SeriesNumber" : "1"
   },
   "ParentStudy" : "350d284e-96867187-900a834d-331e9a21-5649a6b3",
   "Status" : "Unknown",
   "Type" : "Series"
}
```

Instances

The Orthanc Server structures the stored DICOM resources using the **Instance** model of the DICOM standard. Each DICOM resources is associated with an unique identifier:

```
http://localhost:8042/instances/
```

The field **Instances** list all the DICOM studies that are associated with the **Serie**. So, considering the **Serie** above, we would go down in her DICOM hierarchy as follows:

```
[
   "26eb8b88-ff7afef9-e4d3ef85-5fbe799c-3082da71",
   "adf49420-c035cd33-64053e3c-fad3de69-ac9cead4",
   "0e7c5fac-ef55ae7c-ec56e356-74ac42c2-f4f2a32a",
   "89f90986-52a79581-81d5cfef-f0a594b3-f33df53c",
   "3ea91193-d664157d-cdc5e876-fd2b7010-8b9f462f",
   "fddde464-e57f3f55-fc5c0b82-5ee8ea9c-afa8acc4",
   "7aae3a41-051f2909-3e1bc940-1b75e78a-55915f67",
   "a1862117-6f30aebe-b104a727-59eb36c2-bf0f70c4"
]
```

To access a single resource, add its identifier to the URI. You would for instance retrieve the main information about one **Instance** as follows:

```
http://localhost:8042/instances/26eb8b88-ff7afef9-e4d3ef85-5fbe799c-3082da71
```

Here is a possible answer from Orthanc:

```
{
   "AnonymizedFrom" : "2805874b-c3e7a807-e0ecd8ca-fc359751-5ab3b99b",
   "FileSize" : 776288,
   "FileUuid" : "E95F9AA0-B981-4566-A698-3EE5DA175979",
   "ID" : "26eb8b88-ff7afef9-e4d3ef85-5fbe799c-3082da71",
   "IndexInSeries" : 9,
   "MainDicomTags" : {
      "ImagePositionPatient" : "0\\0\\0",
      "InstanceCreationDate" : "20170203",
      "InstanceCreationTime" : "193629",
      "InstanceNumber" : "9",
      "SOPInstanceUID" : "1.2.276.0.7230010.3.1.4.0.895.1491844658.521379"
   },
   "ParentSeries" : "2cec964b-f70d3e7f-e44fb3cb-0ae31f0a-b1878b28",
   "Type" : "Instance"
}
```

Study List

The list of the resources to be sent are given as a JSON file, called studyList.json. In this case, a single DICOM connection is used for each patient. Sample code is available.

The file can be accessed by the following path:

```
src/studyList.json
```

The code is as follows:

```
{
    "studyList": [
        {
            "patientName" : "Anonymized1",
            "patientId" : "5A7A314A-B881-4BEF-91AA-73258CA96675",
            "studyDate" : "20160216",
            "modality" : "US",
            "studyDescription" :"Breast",
            "numImages" : 1,
            "studyId" : "usbreaststudy"
        },
        {
            "patientName" : "Anonymized2",
            "patientId" : "9AF6DD70-51D7-457E-AF53-30CCDC5D7126",
            "studyDate" : "20131101",
            "modality" : "US",
            "studyDescription" :"Breast",
            "numImages" : 6,
            "studyId" : "usmgbreaststudy"
        },
        {
            "patientName" : "Anonymized3",
            "patientId" : "1E60E211-42B6-4556-A1F6-223E85AD38A6",
            "studyDate" : "20161205",
            "modality" : "US",
            "studyDescription" :"Breast",
            "numImages" : 1,
            "studyId" : "us2breaststudy"
        }
    ]
}
```

Studies

The informations of each patient to be read are given as a JSON file for each patient we have. In the further example we show the anonymized5.json file. In this case, a single patient connection is used for each instance (*instanceList*). Sample code is available.

The file can be accessed by the following path:

```
src/studies/anonymized5.json
```

The code is as follows:

```
{
    "patientName" : "Anonymized5",
    "patientId" : "ef05fc61-700f-4f2c-a00e-48cea2443063",
    "studyDate" : "20160210",
    "modality" : "US\MG",
    "studyDescription" :"Breast^Mamografia",
    "numImages" : 10,
    "studyId" : "anonymized5",
    "seriesList" : [
        {
            "seriesDescription": "US",
            "seriesNumber" : "2",
            "instanceList" : [
                {"imageId" : "1.3.51.0.7.4246911301.9636.17487.37142.15423.63151.40726"},
                {"imageId" : "1.3.51.0.7.4246911301.9636.17487.37142.15423.63151.94949"}
            ]
        }
    ]
}
```

Viewport

In *prototype-cornerstone*, an enabled element is a DOM component (e.g. <*div*>) which *prototype-cornerstone* displays an interactive medical image inside of. Internal API call used by *prototype-cornerstone* to add an internal data structure for an enabled element. Internal API functions are not intended to be called by code external to cornerstone.

The calls are done by the *addEnabledElement* function as follows:

```
function addEnabledElement(enabledElement);
```

The *enabledElement* parameter is an internal data structure associated with an enabled element. The function returns nothing.

The *getDefaultViewport* API call is an internal API that returns a viewport object with reasonable defaults given a canvas and image object:

```
function getDefaultViewport(canvas, image);
```

The *canvas* parameter is the canvas element created for an enabled element. The *image* parameter is the image object to be displayed in the canvas. The function returns a *viewport object* with the image centred and scaled to fit all pixels.

The *getDefaultViewportForImage* API call is used to calculate the default Viewport if an image were to be displayed in an enabled element. The default Viewport will set the WW/WC to the default values from the image object and reset the transform so the image can fit on the window.

The calls are done by the *getDefaltViewportForImage* function as follows:

```
function getDefaltViewportForImage(element, image);
```

The *element* parameter is a DOM element that is enabled. The *image* parameter is an image to use to calculate the default viewport. The function returns a *viewport object*.

The *setViewport* API call is used to modify Viewport properties for an enabled element. Calling this method causes the image to be updated to reflect the changes.

The calls are done by the *setViewport* function as follows:

```
function setViewport(element, viewport);
```

Again, the *element* parameter is a DOM element of an enabled element to set the viewport of. The *viewport* parameter, in this case, is an object with viewport properties. The function returns nothing.

A *ToolStateManagers* expose their API through a JavaScript object with several functions so do it. The *prototype-cornerstone* also allows to remove annotations (**Freehand**) supported by this methods. However, there is currently no ability to toggle hide/show of the annotations and measurements. The **Tool Management** page demonstrates how to clear single or multiple tool states.

Clear is only implemented for the *imageId* specific state managers. Therefore, a possible solution is as follows:

```
var toolStateManager = cornerstoneTools.globalImageIdSpecificToolStateManager;
```

Instead of *getElementToolStateManager* before using the clear function.

The *get* function is called by a tool to get tool *data* for a specific *element* and *toolType*:

```
function get(element, toolType);
```

One more time, the *element* parameter is a DOM element to be enabled. The *toolType* is an unique string identifying for this tool data. This needs to be unique to avoid conflicts with other tools so consider using a GUID, project name or other strings as part of the *toolType*. The function returns a JavaScript object containing the tool data or undefined if there is no associated tool data.

The *set* function is called by a tool to set tool *data* for a specific *element* and *toolType*:

```
function set(element, toolType, data);
```

Again, the *element* parameter is a DOM element to be enabled. Also, The *toolType* is an unique string identifying for this tool data. This needs to be unique to avoid conflicts with other tools so consider using a GUID, project name or other strings as part of the *toolType*. The function returns a JavaScript object containing the tool data or undefined if there is no associated tool data. But this time, we have a *data* parameter with the purpose of having a JavaScript object containing the tool data. The function returns nothing.

The *loadImage* API call is used to initiate a load of an image given an *imageId*. Our *prototype-cornerstone* first checks the cache to see if this *imageId* was previously loaded. If the *imageId* is not in the cache, *prototype-cornerstone* attempts to locate a suitable *ImageLoader* for the *imageId* and requests the image to be loaded. Since image loading is an asynchronous operation, this function returns a *Promise* which will resolve to an image object on success. Typical workflow is to call *displayImage()* after a successful load. Note that this function will attempt to locate the *imageId* in the cache but it will not add loaded images to the cache.

The use of *loadAndCacheImage()* if you would like to cache the loaded images is important for this and you should do as follows:

```
function loadImage(imageId);
```

The *imageId* parameter is the *imageId* to load. The function returns a promise which resolves to an Image object once loaded.

The path for our loadStudy.js is as follows:

```
src/js/loadStudy.js
```

You can always edit the *imageId* and *image.imageId* conditions and variables on this file.

To conclude, each enabled element has a viewport which describes how the image should be rendered. The viewport for an enabled element can be obtained via the *getViewport()* function and set using the *setViewport()* function.

Additional Information

Authors

— Francisco Maria Calisto [GitHub] [ResearchGate] [LinkedIn] [IST] [INESC-ID] [ISR]
E-Mail: francisco.calisto@tecnico.ulisboa.pt