Make A Creative Application!

Draft version: May 9, 2025

https://www.bighillsoft.com

Before We Start

MACA Suite Proto, simply Maca Proto or just Maca, is a prototype version of the MACA Suite that will be released later. Since this is a prototype, all front and back-end features are subject to change in the next release. Some stated or unstated features are not implemented, as you may notice while playing with them, and will be implemented in the future prototype release. Still, there are many unstated bugs, any unknown glitches could be found as well. So please feel free to let us know at maca@bighillsoft.com about those for the fix or patch. It is minor, but the informative popup and alert are still somewhat inconsistent and need to be improved. Lastly, the UI style and layout is not fixed yet, and it looks awkward but will be enough to test with. Let's make a creative application!

Introducing MACA

MACA Suite Proto developed with .NET MAUI[1] is a combination of Macadamia and Macaron; Macadamia is a front-end UI and Macaron is a back-end application, i.e., a server. The name MACA stands for *Multi-threaded and Asynchronously Communicating Application,* which is a service that you will develop through Macadamia and run on Macaron. As the ambiguous term "service" appears here, you will be a bit confused by its broad meanings. Throughout this tutorial, this will be explained in detail with examples.

Macadamia

Macadamia, or simply DAMIA or Damia, is a *Development And Management Integrated Application* that you will use most of the time. As the name suggests, Macadamia is divided into two main workstations: Development and Management. You will use Development workstation to create services and Management workstation to maintain the services running on the Macaron server. Since the first step to using Maca is to develop services, you will see the Development workstation when you start Damia for the first time. We will cover how to switch or set the default startup workstation later.

Macaron

Once you have developed services that don't run by themselves, a server is required to load and run them. Macaron, or simply Ron, was built on an ASP.NET[2] server. This back-end application needs to be installed on the same machine where Damia was installed or on a remote machine that may not have Damia. It is designed to interact with Damia within a network using the gRPC[3] protocol so that it receives requests such as start or stop services that are loaded on it. As Damia can request current or historical data about the services, it stores important transaction data in an embedded database, SQLite[4].

Basics of MACA development

Models

Before building services, it is necessary to understand how they will be structured. In order to make the service more clear, a hierarchical tree structure with an identifiable model concept is used. A typical conceptual example of a model is each name of your

[1] https://dotnet.microsoft.com/en-us/apps/maui
[2] https://dotnet.microsoft.com/en-us/apps/aspnet
[3] https://grpc.io/docs/languages/csharp/ and https://github.com/grpc/grpc-dotnet
[4] https://www.sqlite.org/

company's internal organization, and a popular example in action is a folder and file in Windows® File Explorer. As Macaron requires a client's machine or device to run services, structuring the client's organization and machine along with the project in a top-down way will be a key point to making services stand out. See below for key models.

Domain

Domain model, or just Domain, is the highest-level model that describes your client. You can consider this as a company, hospital, website, or whatever you will work for. So, this is the first model you have to create when you start building Services. The sample value could be "WA Healthcare" or "Seattle Clinic," depending on the size of the client.

Instance

Instance model, or just Instance, is a machine or device on which the Macaron server will be installed. This will be located under the Domain. The sample value could be "PC1" or "Device1." As a company can have multiple computers, Domain can have multiple Instances.

Note: Maca Proto doesn't support sub-domain concepts like departments, teams, or buildings under the Domain. You may set the Instance name as a combination of sub-domain and Instance, like "Seattle PC1" or "Bellevue PC1," for example.

Project

Project model, or just Project, is a set of services that run on the Instance. You may set this as an actual project name or your own project name that describes the objective of the services, like "HL7 order project." An Instance can have multiple projects.

ServiceGroup

ServiceGroup model, or just ServiceGroup or simply Group, is a group of services for their own purposes under the project. Likewise, a Project can have multiple ServiceGroups. In addition, each Project will have a Testers group that will be used as a location for all test services for the Project.

MacaronService

MacaronService model, or just Service, is a model that runs on the Macaron server. This standalone model is a set of user-defined and built-in logics. As you develop the services with C#, they will require a .NET environment. And the services need to be loaded on the Macaron server. For this reason, a service is called "Macaron Service," and its file will have an extension ".ron," which stands for runs on .NET. Throughout this tutorial, the service file is sometimes called a "ron file," alternatively. You can use Damia's Management workstation to load and maintain services on the Macaron server. Based on its execution modes, the code will run in a timely manner. Maca also provides built-in service types that interact with a specific service or application, so that a user can just focus on the logic itself. A service can be a part of Project or ServiceGroup based on its purpose or functionalities.

Note: Except for the MacaronService, all other models also called group models since they can have sub-models. Please note that the ServiceGroup is also a group and can contain only MacaronService. Throughout this tutorial, sub-models will be called children.

Mapping models to physical files

As you will see later, Maca uses its own file extensions to open, save, or run models. Based on the models stated above, each model will have its own file extension, as described in Table1.

Model	UI icon	File extension	File examples
Domain		.domain	WA Healthcare.domain
Instance[5]		.inst	Seattle PC1.inst
Project		.proj	HL7 order project.proj
ServiceGroup		.group	Testers.group
MacaronService[6]		.ron	DICOM Reader.ron

[5] As an instance could be a different type of machine or device, the icon will not be the same as one in Table 1. Instead of the PC icon, other icons, such as tablet ( ) or even chip ( ), can be assigned to the instance.

[6] Along with this regular service, there are supporting services such as Tester and Template. They will have their own icon as well.

Table 1.Model and File Mapping

Below Figure 1 is an implementation of sample projects for a hypothetical client, WA Healthcare. We will go through this structure by implementing each model one by one later.

Figure 1.Sample projects

Note: Maca uses a hierarchical file structure to identify each model. So, it is not recommended to manually modify a file's content or extension, which may cause unexpected crash or result. Or even relocating file or folder to a different location may break the project's consistency that was maintained by Maca.

Supportability

In order to build projects, one thing that must be checked is the environment that Maca runs on. Please take a look at the following system requirements.

Operating Systems

The original idea of Maca was to run on most of the operating systems in the market. Since the Maca prototype was built on a Windows 10 machine and ported to Windows 11, Windows 11 machines will work best. In addition, as briefly mentioned in the above section, Macaron was built on ASP.NET, which doesn't support Android, so that the actual release of Android version might be later than the other OS versions. Please check Table 2 to see what will be available.

OS	Macadamia	Macaron	Releases
Windows	Windows 10+	Windows 10+	Prototype
iOS/Mac	N/A	N/A	TBD
Linux	N/A	N/A	Targeted next prototype
Android	N/A	N/A	TBD
Tizen	N/A	N/A	TBD

Table 2.Maca proto supporting OS

.NET Runtimes

Maca proto is currently targeted for .NET 9. Next release will support .NET 10+.

Macadamia

Maca is designed to work in any environment and field of interest, but it initially started with a healthcare project. Because of that, some icons are related to the laboratory, as you may notice. Since it is still a prototype version, not all environments are available for now. For this reason, this tutorial is focused on the healthcare project on Windows environment. Although this may limit its applicable fields, you can extend it to any project that you will work on once you finish it. You might be new to the medical field, but we assume that you have some experience in implementing client's requirements because this tutorial requires some programming skills. If you feel comfortable with this, let's start building the first project.

Build a project: Scenario1

Directions

You are assigned to a project from a hypothetical client, WA Healthcare that has multiple clinics in Washington State, to build an interface application that communicates with your client's application, such as an EMR[7] or interface engine. One key role is to create an application with Maca that runs on a Windows machine that resides on their premises. Your application will use a variety of media, like DICOM[8] files or HL7[9] files that were generated from multiple devices or applications on the client side.

The client may not provide a test environment to test your application from the start of the project. However, they may provide sample media that your application can test with. Or, they assume that you are already familiar with the media and provide brief information without actual files.

Objectives

For simplicity, your application will do the following;

- Reads exported DICOM files and sends HL7 order messages to the client's application.

Note: In this tutorial, no specific medical field or situation is targeted. A real-world scenario could be different from this sample hypothetic scenario.

With this guideline, let's build an application.

 Development workstation

Once you install Maca and run Macadamia, initial workstation titled “Develop Instances” will show up. This is the default startup workstation whenever you start Damia. The Development Workstation is divided into 3 main areas: **Model Explorer** (left), **Model Viewer** (center), and **Templates** (right). Since the Model Explorer shows the projects, it is also called Project Explorer. We will go over each area later.

[7] https://digital.ahrq.gov/electronic-medical-record-systems
[8] DICOM stands for Digital Imaging and Communications in Medicine. See https://www.dicomstandard.org/about for further information
[9] HL7 stands for Health Level Seven. See https://blog.hl7.org/author/health-level-seven and https://www.hl7.org/index.cfm for further information.

Figure 2.Development workstation

Although the default workstation is Development, you can switch to the other workstation or set the default startup workstation as you prefer.

Switching workstations

In order to switch workstations, in this case to Management, click the flyout menu button (a.k.a., a hamburger button) on the top-left of the application and click Management menu. This will switch the current workstation to Management.

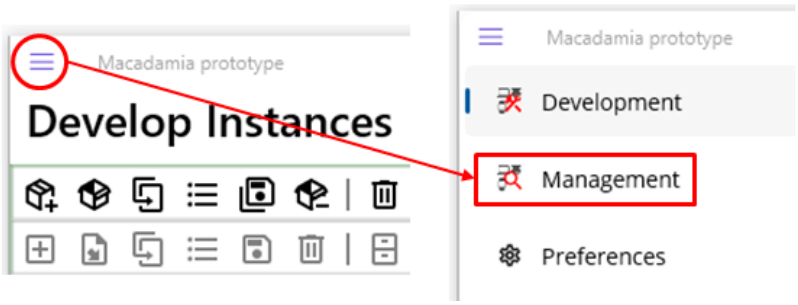


Figure 3.Workstation switching

Set startup workstation

Although you switched a workstation to Management, you still see the Development workstation once you close and restart Damia. In order to see Management all the time when Dmia starts, you need to set that on the Preferences page. Figure 4 shows how to set up a startup workstation.

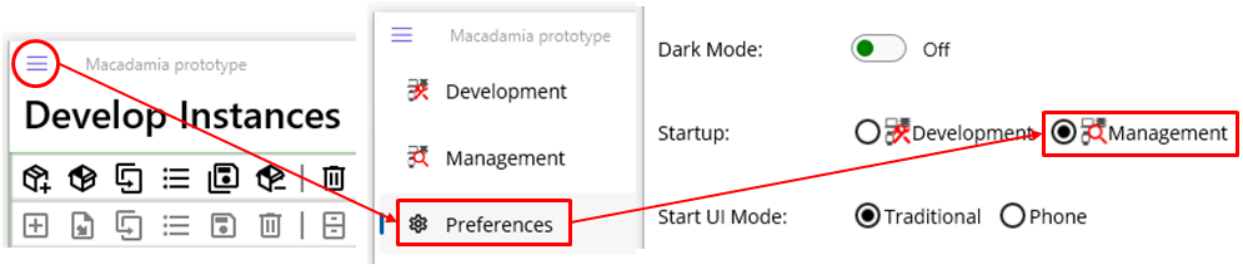


Figure 4.Startup workstation setup

Create Instance

As briefly mentioned several times earlier, Maca uses a hierarchical structure to build services. So, creating models in a top-down way is the baseline of development. So, the first step in building a project is to create a Domain model.

Before creating a Domain, let's take a look at the Instance toolbar below.

Figure 5.Instance toolbar

Conceptually, Domain can have multiple Instances. But Maca, specifically Macadamia and Macaron, can load only **one Instance at a time** because it is designed to focus on one specific machine or device that Macaron services will run on. Loading multiple Instances into Maca is not supported. And the Instance toolbar is set to open or create only one instance at a time. So, once you load an Instance, you can assume that your machine is now the client's machine. However, as you will notice later, there is no Domain-related button in the Instance toolbar as the name itself describes. Instead of having that button in it or putting it somewhere else, the Instance toolbar contains Domain-related functions. This will be clear when you click the Create Instance ( ) button. Please click the Create Instance button. When you hover the mouse cursor over the button, a corresponding hint will pop up.

Figure 6.Create Instance button

Once you click the button, the Create Instance popup will show up.

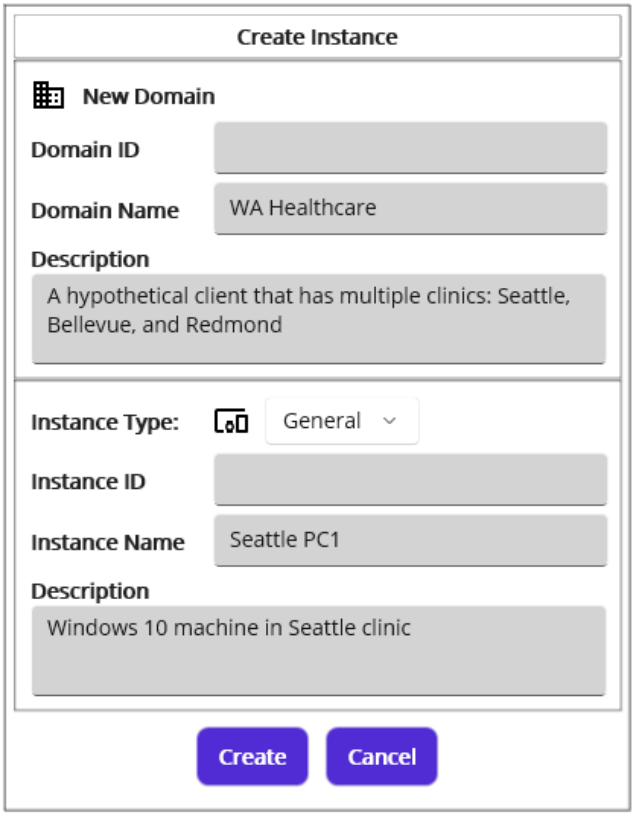


Figure 7.Create Instance popup

The Create Instance popup is composed of two main sections: Domain and Instance. Since we haven't created any Domains yet and our first step was to create a Domain, we need to provide new Domain information. As you see sample values populated with scenario information in Figure 7, you can type values about your client.

In the Domain section, Domain Name is a required field. Among the optional fields, Domain ID is a custom value that might be assigned by you or the client and is not auto-generated by Maca.

Followed by Domain, you can type Instance-related information. Since the Instance is equivalent to a machine or a device, you can assign its type by picking one of the items in the Instance Type Picker[10] shown in Figure 8.

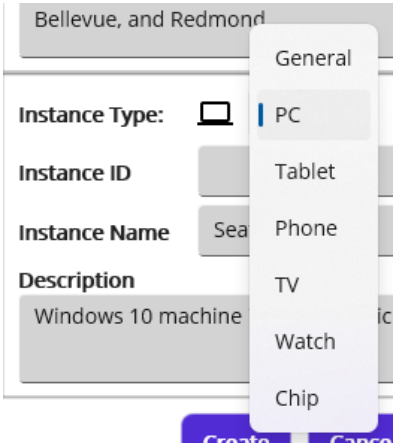


Figure 8.Instance Types

Like Domain, Instance Name is a required field, and the optional field Instance ID can be assigned a custom value.

[10] Usually, this is called a drop-down list. Throughout this tutorial, we call it Picker.

Note: It is important to note that the Instance type in the Create Instance popup is closely related to the Macaron server that will be installed on a client machine or device that may have a different OS. As you see in Table 2, the prototype version of Macaron runs on Windows 10 only for now. If you choose other types, such as TV that might have Android OS, it is assumed that the services are designed to run on Android OS, which is not supported by the prototype Macaron server. Still, there is no restriction on which type to choose. However, the PC type will be used for both this tutorial and Maca Proto in most cases to avoid any confusion.

Once you provide the required values, click the Create button to create a Domain and Instance.

In a model explorer, created Domain and Instance will be shown in Figure 9 below. Please note that the Model Explorer is a container that shows models as a hierarchical tree structure and is sometimes called Project Explorer as this shows projects.

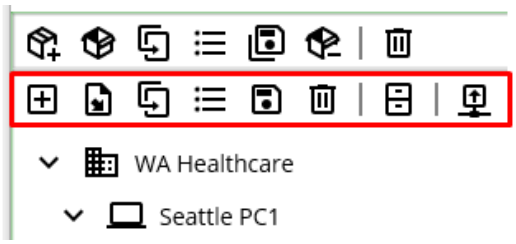


Figure 9.Created Domain and Instance models under the Project toolbar

Additionally, the Project toolbar will be enabled right below the Instance toolbar. This means that it is ready to create a project that belongs to the Seattle PC1 Instance.

Source location

Once you create a Domain along with an Instance, those models will be saved under the following default location.

C:\Maca\Damia\Works

Note: Macadamia doesn't support saving the model outside of this default location when creating an Instance. But still, it can open and save Instances that were manually stored outside of this location.

The created Domain and Instance will be structured as shown below in Figure 10.

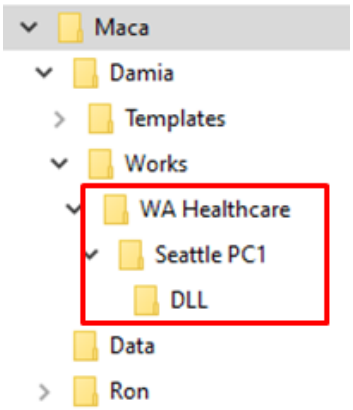


Figure 10.Created Domain and Instance files

As you see in Figure 10, there is a subfolder named DLL. That will be used as a central location to store 3rd-party DLL[11] files that can be used for the services under the Instance. Although this is part of the Instance, it won't be shown in a model explorer because it is just a folder for DLL files. The DLL folder will be covered more in the Service Creation section.

[11] https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries

Create Project

As we have created a base location to store our services, it is time to create a project assigned to you. Click the Create Project (⊞) button to open the Create Project popup.

Figure 11.Create Project button

Following is the Create Project popup with populated values. Please note that the Description is the objectives defined in Scenario 1. Like an Instance popup, Project Name is required. Click the Create button.

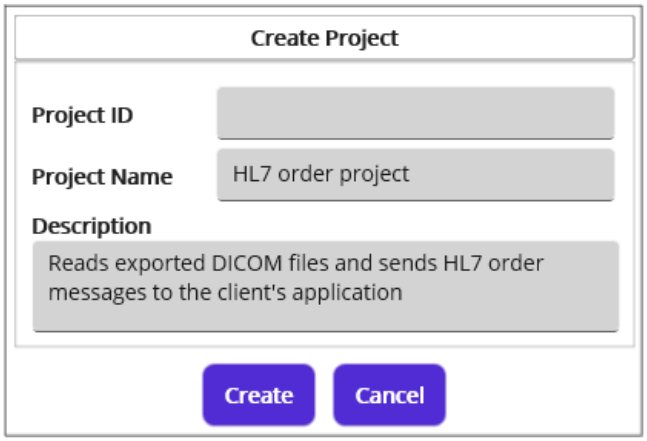


Figure 12.Create Project popup

In a model explorer, the HL7 order project that you just created will be added under the Seattle PC1 Instance, as shown in Figure 13.

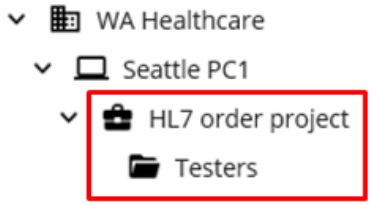


Figure 13.Created HL7 order project with a default Testers group

Like Instance, Project will have a default sub-folder named Testers which is also a ServiceGroup to store all tester type services. Whenever you create a new project, a Testers folder will be created as well.

Create ServiceGroup

For now, we may skip this part and use Project as a main location to store services. We can go over this topic later.

Create MacaronService

In order to create a service, it is required to select Project or ServiceGroup node [12] in a model explorer to define where the new service will belong. Otherwise, the Create Project popup will be shown when you click the Create Project Model button. We do have a Testers group, but it is for tester-type services, so we select the "HL7 order project" node instead to add a service. Then click the Create Project Model button to open Create Group or Service popup.

Note: The Testers folder was designed to have services for testing purpose, i.e., not for production level services. So, the services exist outside the Testes are called required or necessary services.

[12] Each model inside a model explorer is called a node, a Domain node or a Service node for example.

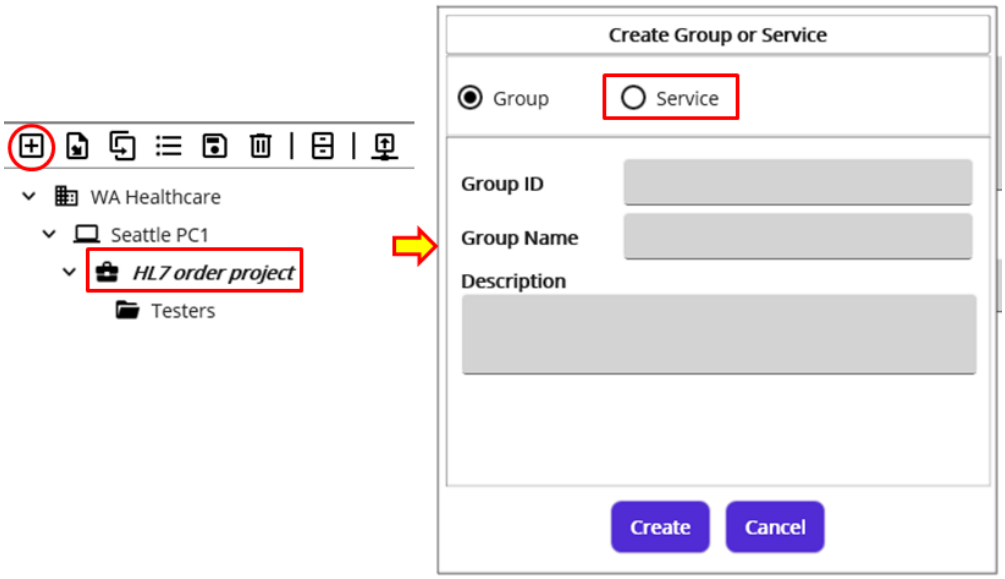


Figure 14.Open Create Group or Service popup

As Figure 14 shows, the Create Group or Service popup shows a group-adding section first since the Group radio button is selected. We have to create a service, so click the Service radio button to switch to the service-adding section.

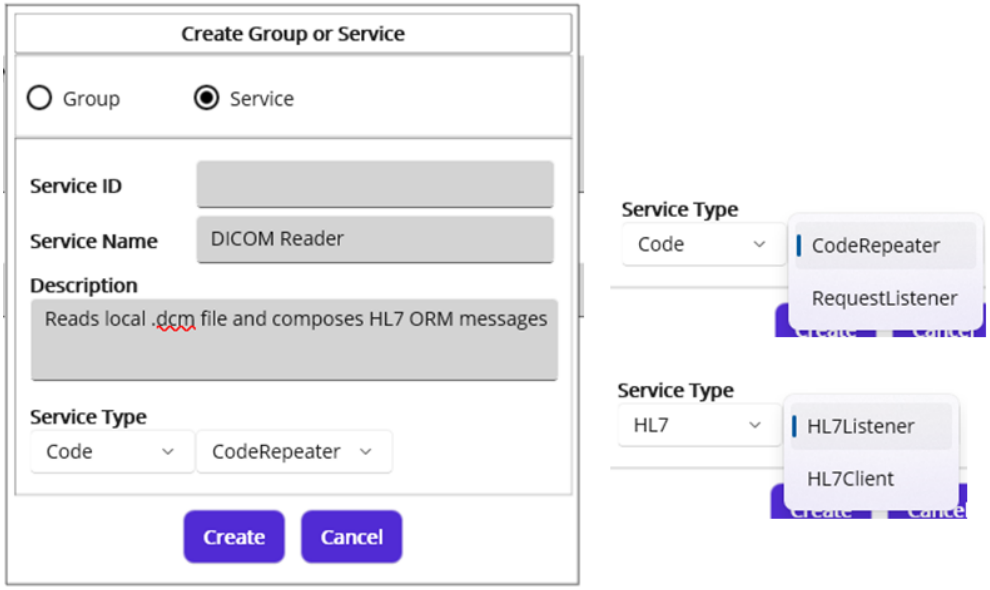


Figure 15.Service-adding section and Service Types

On a service-adding section, you can type the base information that defines this service. Then choose an appropriate service type from Service Type Pickers. As Figure 15 shows, there are two main service types: Code and HL7.

- **Code** type is designed to run user-written C# code.
  - **CodeRepeater** executes code based on a time interval, like every 10 minutes or once a day.
  - **RequestListener** executes code when a request is received.
- **HL7** type is designed to communicate with external or internal HL7 services by using user-written C# code. Its services are connection-based, so they require a TCP/IP-based client or listener that implements MLLP (Minimal Lower Layer Protocol) internally.
  - **HL7Listener** receives HL7 messages and requires an external client or HL7Client Service.
  - **HL7Client** sends HL7 messages and requires an external listener or HL7Listener Service.

The service will read a DICOM file that has a .dcm extension and compose HL7 messages, so the CodeRepeater is a good candidate to meet these requirements. Select Code type and CodeRepeater service. Then click the Create button.

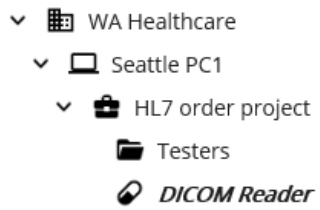


Figure 16.DICOM Reader service belongs to the Project

As you see the DICOM Reader service in Figure 16, it belongs to the HL7 order project, i.e., HL7 order project has two children: the Testers ServiceGroup and the DICOM Reader MacaronService. Figure 17 shows the difference between the project service and the group service.

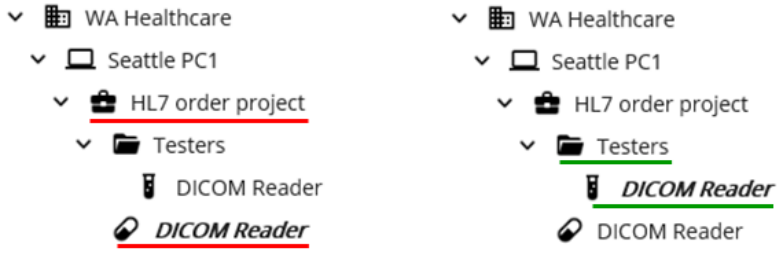


Figure 17.Project Services (left) vs Group services (right)

Note: Unlike traditional Windows File Explorer's left-side tree view, the model explorer doesn't always maintain a folder-style node since the service node, in this case DICOM Reader service, should be displayed as well. You can still categorize the service and put it in a specific group to organize services based on their objectives or types in a traditional way. But if you think the service is more relevant to the project itself, you can put the service under the Project. However, if you think that is visually confusing or unnecessary, you can always create a group first and then put services there.

Now let's dig into the service development.

Model Viewer: Service

In order to achieve a complex goal, a service is composed of pre-defined, configurable contexts. You can manipulate those contexts using the Model Viewer. And each context is represented as an editable view, such as Detail View, Deploy View, and Steps View, in a service editor. Since Service Editor shows only one context at a time, navigating each context is required to edit them. Please note that the Macaron server converts those configurations to executable code when it deploys the service.

Select a MacaronService node, in this case the DICOM Reader, and the service model will show up in the model viewer, as shown in Figure 18.

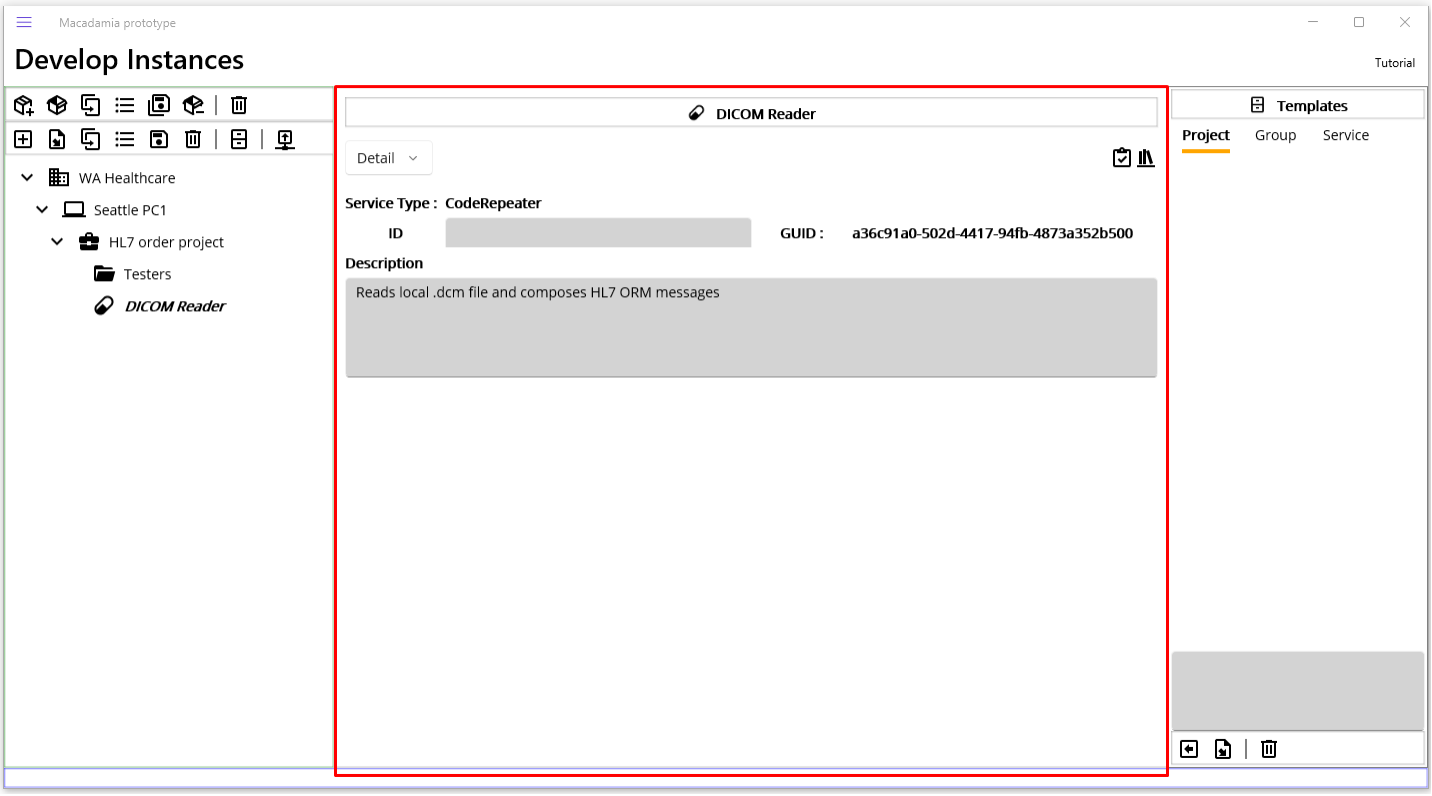


Figure 18.Selected Service Model in a Model Viewer

In case of the MacaronService, the model viewer shows two areas: the **Service Toolbar** and the **Service Editor**. For other models, such as Domains or Projects, the model viewer will show a different View, which will be covered later.

Service Toolbar

At the top of the viewer, there is a Service Toolbar that contains the **Service Navigator**, the **Verify Service ( ) button**, and the **Manage Service Libraries ( ) button**.

Figure 19.Service Toolbar

Service Navigator

This allows you to move around each context view of the service. Each context can be called a combination of its own name and view, such as Detail View, Deploy View, or Steps View. Or you may simply call it by its name like Detail or Deploy.

Figure 20.Service Navigator: Detail, Settings, and Execution

As you pick one of the items (Detail, Settings, and Execution) of the main Picker, sub-Picker shows up accordingly, like Figure 20 example. Based on the selection, corresponding view shows up in the Service Editor under the Service Toolbar.

Note: Since this is a prototype version, more options in the Service Navigator could be added in the next release.

Verify Service

This verifies the service was configured correctly and the C# code was compiled successfully.

Manage Service Library

This manages all DLL-related configurations. This will be covered later.

Service Editor

When you select one of the items in a Service Navigator, the corresponding context view will show up in a Service Editor. Figure 21 shows the Config view in a Service Editor when you navigate to Settings>Config.

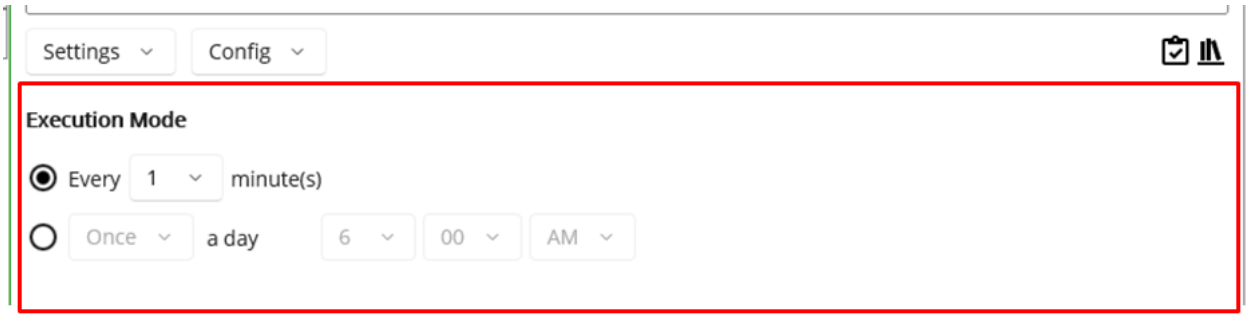


Figure 21.Service Editor

CodeRepeater

As we choose the CodeRepeater service to implement the requirements, it is necessary to check what configurable features each context has. Using Service Navigator, let's go to each context to set appropriate values.

Edit Detail

When you create a service, you will provide basic information about the service. If you want to modify or add extra information to the service, such as Detail or even Service ID, type here. For example, you may add short instructions, such as a step-by-step guideline or what DLL is required, in the Detail section.

Edit Config

This is one of the key parts of the CodeRepeater because it defines when Macaron executes the code you wrote in the Steps View. As Figure 21 shows, 1 minute is the default interval. Optionally, you can select one of the intervals of 1, 5, 10, 30, or 60 minutes. For this project, we can change the default value to **5 minute**.

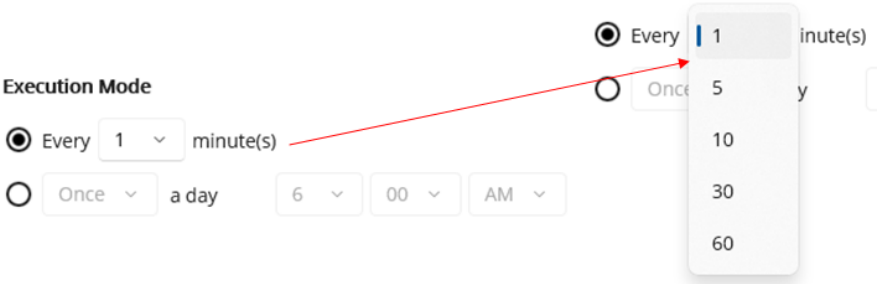


Figure 22.Set Execution Mode

Note: Maca proto does not provide second(s) interval.

Alternatively, you can set the execution time to once a day or twice a day if that is more relevant to the requirement.

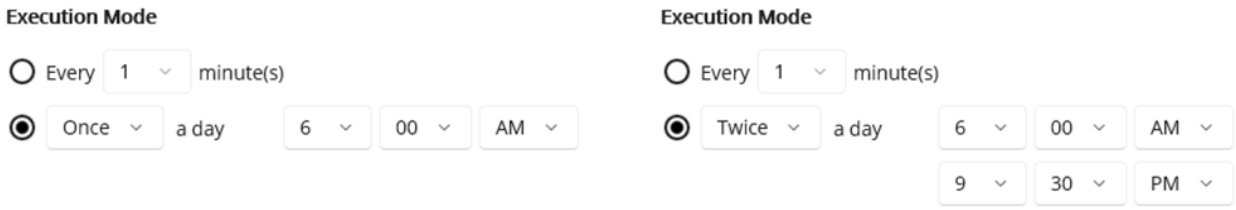


Figure 23.Execute Once (left) or Twice (right) a day

Edit Deploy

Settings Deploy

The C# code here will be executed one time when the service is deployed on the Macaron server. If there is any precondition before the service starts, this is a good place to add that logic. Deployment will be explained in the Management workstation section.

For example, if the service requires a certain location, such as a local folder, put the folder creation code here instead of the Steps view that runs every minute. The following will run once when Macaron or you deploy the DICOM Reader service.

Note: The following examples are part of the C# syntax that you may already know. If you are new to or not familiar with that, please search online tutorials or visit the Microsoft C# documentation site[13].

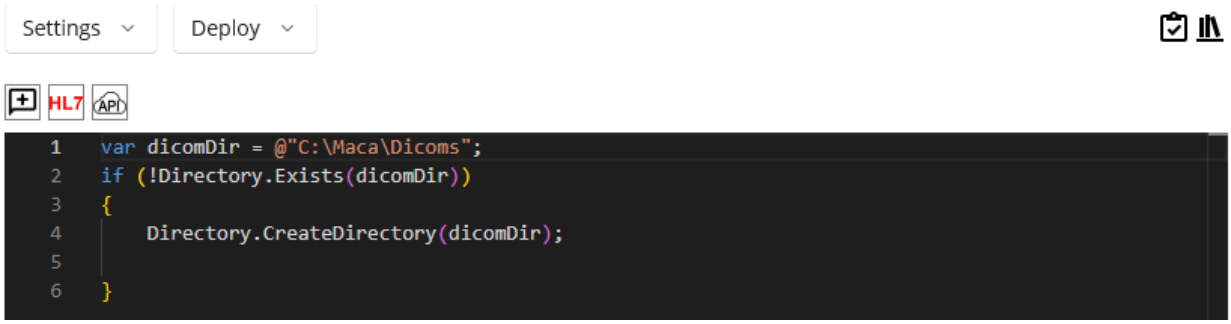


Figure 24.Deploy code editor

Note: As you see in Figure 24, Maca proto doesn't implement an IDE-style code editor or autocomplete. All code that you write in each code editor will be shown as simple multi-line text, and squiggly underlines might be seen due to the spellcheck. Those features will be added in a later release or in a beta version, hopefully. For now, the Verify Service will proofread the code in a service.

Edit Undeploy

Settings Undeploy

This is designed to be executed one time when the service is un-deployed from the Macaron server. Undeployment will be explained in the Management workstation section.

Note: Undeploy is not implemented in Maca proto.

Edit Filter

Execution Filter

Note: CodeRepeater doesn't include Filter. Use Execution > Steps instead to add the necessary filtering code.

Edit Steps

Execution Steps

[13] https://learn.microsoft.com/en-us/dotnet/csharp/

Along with the Edit Config, this is the core part of the service that you will actually implement in C# code. Based on scenario 1's objectives, we can divide the DICOM Reader's core procedure into the following key steps;

1. Finds DICOM files
2. Parses and extracts DICOM file's internal data
3. Composes ORM^O01 message using extracted data.
4. Sends the message to the ORM Sender service.

Finds DICOM files

The DICOM file has the ".dcm" extension. So, reading that file in the designated location is required. You can implement that logic with loop-style code to keep checking if there's a new file. But please remember that the DICOM Reader service, more specifically the code in the Steps view, runs every minute as we configured it in the Config view. That being said, you can focus on the logic itself without worrying about how to repeat it.

DICOM

A Digital Imaging and Communication in Medicine[14] (DICOM) file is required to test the DICOM Reader. The National Electrical Manufacturers Association[15] (NEMA) is also a great resource to look at for the details. This tutorial does not provide sample DICOM file but you can search web for sample DICOM files, so we assume that you already have sample DICOM files. For those who use FTP client application, such as FileZilla (https://filezilla-project.org/), we recommnd accessing **NEMA's FTP server** (ftp://medical.nema.org/medical/Dicom/Multiframe/CT/) to get sample DICOM files because this tutorial uses them as well. Once you obtain some sample dcm files, extra application to view internal data will be helpful to cross-check the values. Since some sample dcm files you have may not have what this sample scenario requires, your code is required to catch that exception to prevent an error. If you work with medical devices from different manufacturers in the future, you will notice that some Tags are filled with a different value (format) or even do not exist, which is totally fine for their own purposes. So, it is required to check internal Tags using **a DICOM Viewer**, such as Microdicom DICOM Viewer[16], all the time when you handle the DICOM files. Along with fo-dicom website, NEMA's website including *Registry of DICOM Data Elements* (https://dicom.nema.org/dicom/2013/output/chtml/part06/chapter_6.html) provides a detail of the DICOM Tags. Figure 25 shows parts of the Tags captured from the sample DICOM file for reference.

DICOM Tags

Search...

(Group,Element)	TAG Description	Value
(0008,9205)	Pixel Presentation	MONOCHROME
(0008,9206)	Volumetric Properties	VOLUME
(0008,9207)	Volume Based Calculation Techniq...	NONE
(0010,0010)	Patient Name	CardiacCalcium PostOperative
(0010,0020)	Patient ID	0001
(0010,0030)	Patient Birth Date	19500704
(0010,0040)	Patient Sex	M
(0010,1010)	Patient Age	052Y
(0010,1020)	Patient Size	1.6
(0010,1030)	Patient Weight	75
(0018,1000)	Device Serial Number	123456

Figure 25.Sample DICOM Tags with hypothetical data

Parses DICOM file's internal data

Every file has its own unique properties and content, and most of the files will not be parsed with default .NET libraries. The DICOM file is one of them and requires an extra DLL to be parsed. A typical example is extracting a patient name and ID along with

[14] https://www.dicomstandard.org/
[15] https://www.nema.org/
[16] https://www.microdicom.com/dicom-viewer.html

embedded images, such as CT or MRI. There are a couple of libraries that do the job, and we will use fo-dicom[17] as an example. Then we face the question about how to add those DLLs to the project and where to find helpful ones.

NuGet[18]

One common way to obtain DLL files is to search NuGet and download a package. The most popular example is Microsoft® Visual Studio[19], which searches for DLLs and installs them using the embedded NuGet package manager. Similarly, but a little bit differently, Damia uses NuGet as the main source of the DLL.

As we are still on the Model Viewer, if not, please click service model in the model explorer, click the Manage DLL ( ) button.

Figure 26.Manage DLL button

Once you click the Manage DLL button, the Manage Service Libraries popup will show up.

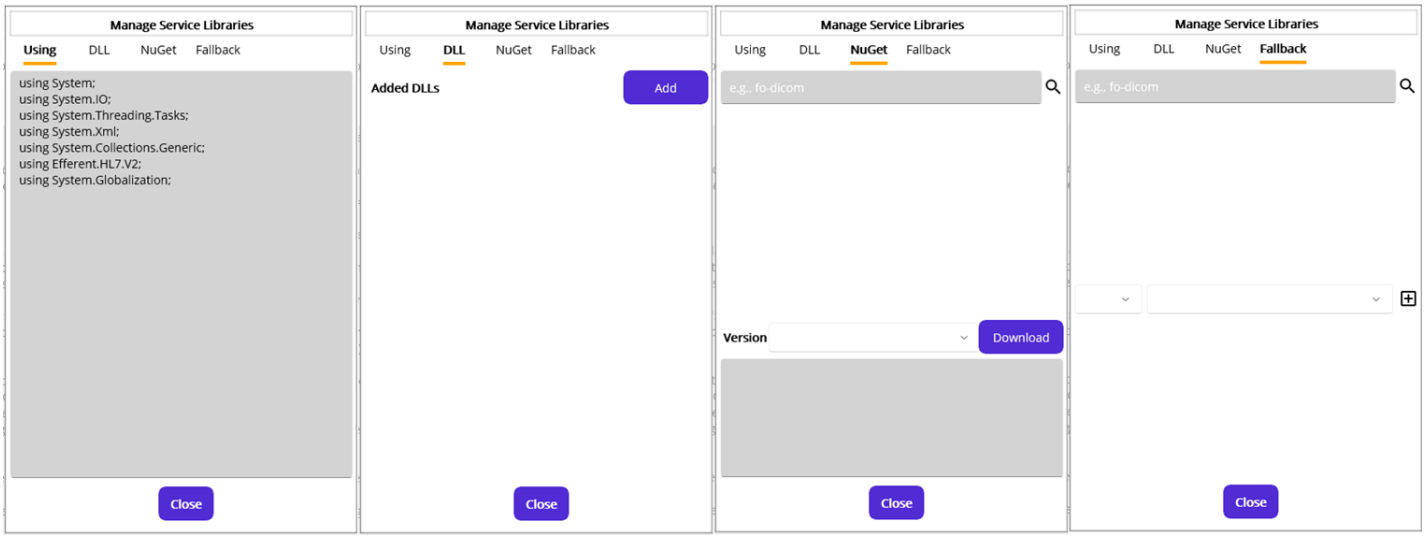


Figure 27.All tabs in the Manage Service Libraries popup

As Figure 27 shows, there are 4 tabs that we can configure. Firstly, we need to search for the “fo-dicom” package ID, so click the NuGet tab.

NuGet Tab: Search and download DLLs

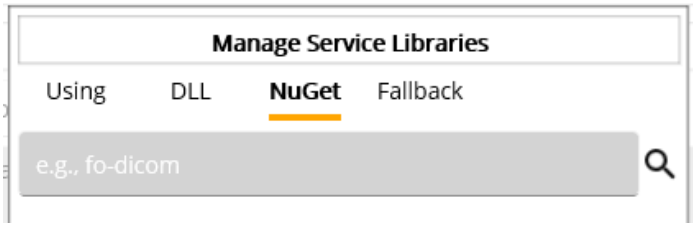


Figure 28.NuGet package ID search field

[17] https://github.com/fo-dicom/fo-dicom
[18] https://www.nuget.org/
[19] https://visualstudio.microsoft.com/

As the placeholder value “e.g., fo-dicom” shows, you can type a search term in the search field. Please note that “fo-dicom” is a package ID that was registered to NuGet. You can type any search term, but the result will be relevant to the registered package ID. In our case, type “fo-dicom” and click the magnifier button, i.e., a search button. And the result will be shown in Figure 29.

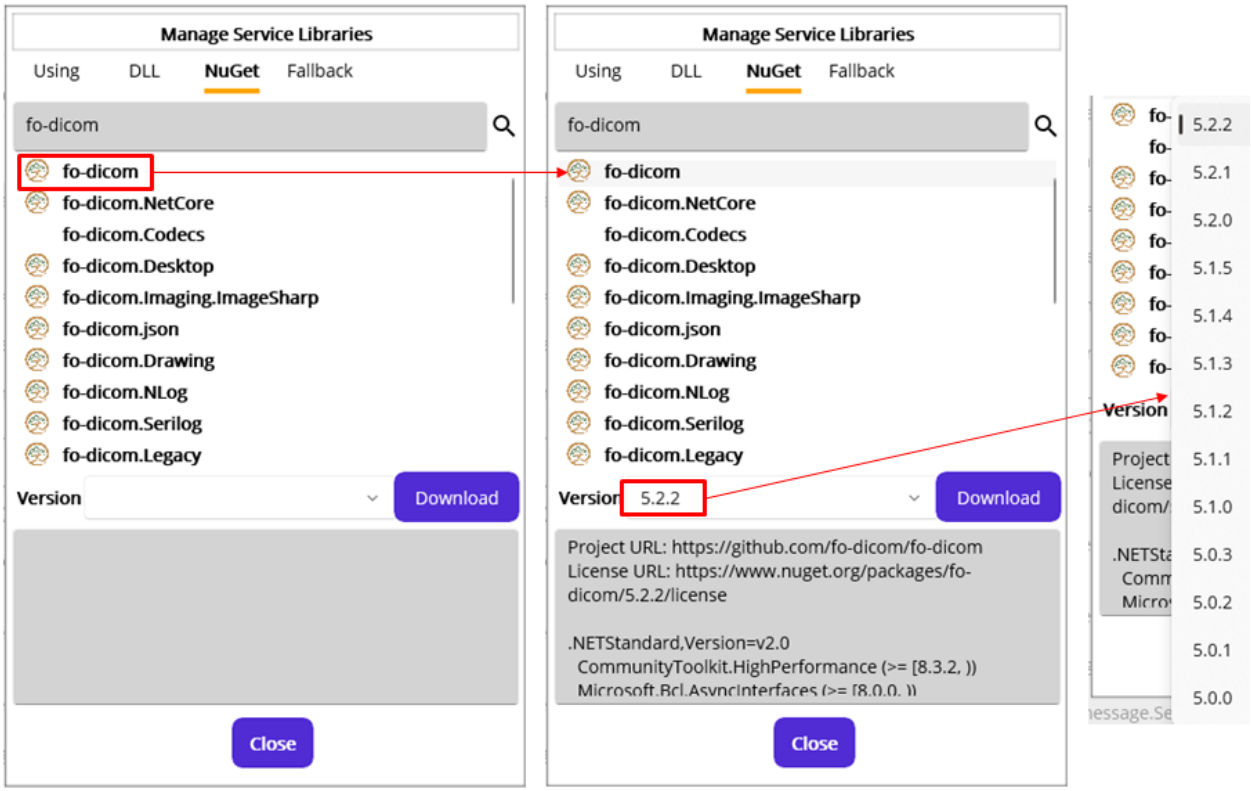


Figure 29.DLL search result with “fo-dicom”

From the results, you can click on any of the packages to see if what you are looking for is in the list. In our case, the top one from the result is the one. Click on the name of the package, and the version list and details will be shown below. Please note that when you click the result package, the latest version will always be selected, like 5.2.2 in Figure 29. But you can select any of the previous versions that fit your service.

Note: The number of the search result is set to 20.

Since we found the right package for the service, click the Download button next to the version. That will download the package file and extract to the following location.

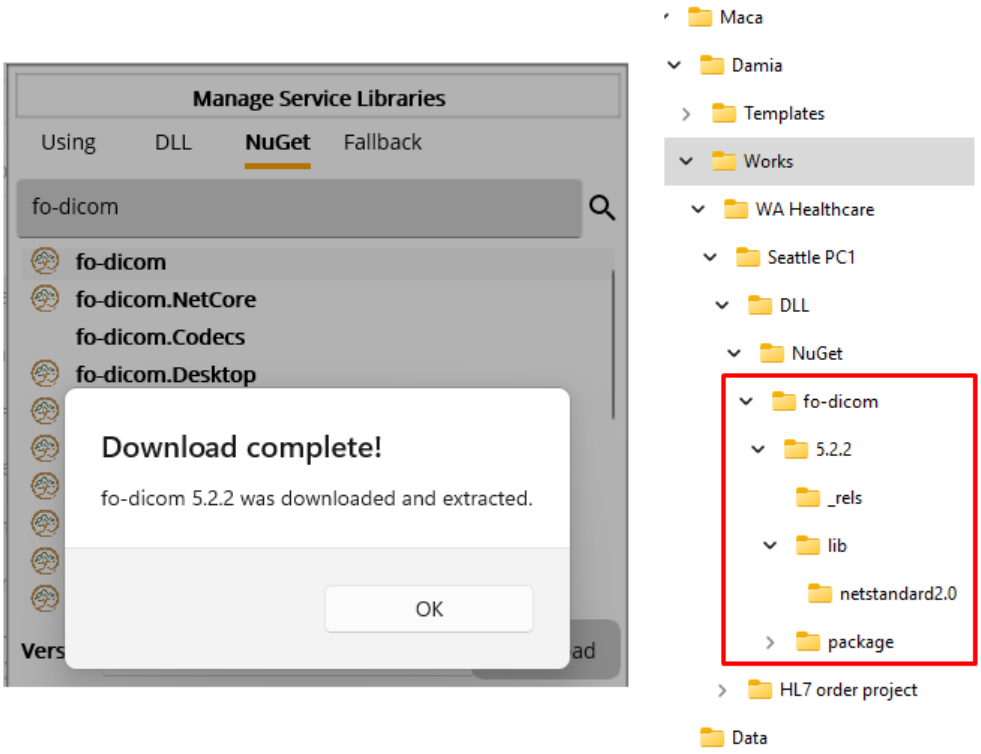


Figure 30.NuGet package download

In the same way, you can search for other packages and download them to the location under the DLL folder. Before adding download dll files to the Instance, let's write sample C# code that relates to the other tabs.

Using Tab: Declare namespaces for Steps

You can declare using directives, i.e., namespaces, here in order to use .NET framework or 3rd-party libraries. You can add those libraries through the DLL tab that we will cover later. There are pre-loaded namespaces, but you still have to add one that is not declared. For example, if you want to use StringBuilder, you have to add "using System.Text;" otherwise, you will see the error message shown in Figure 31 when you click the Verify Service button in the Service Toolbar.

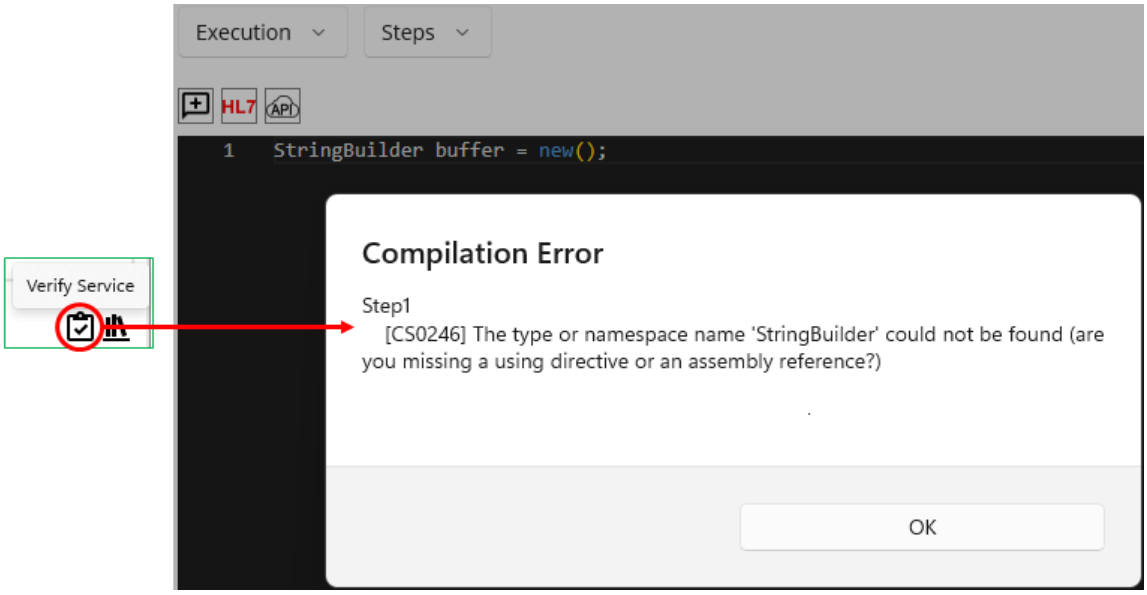


Figure 31.Reference Error

Along with pre-loaded namespaces, you can add the required namespace in the Using tab, like in Figure 32.

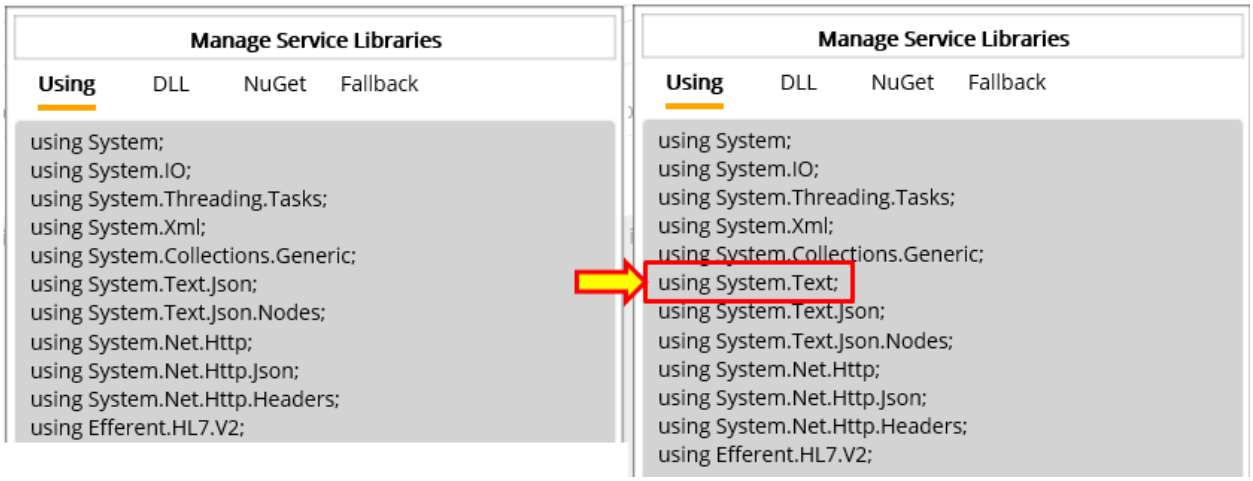


Figure 32.Add Using Namespace for StringBuilder

In our scenario, if we use fo-dicom's method in the Steps and don't put the required namespace in the Using tab, you will see an error message like shown in Figure 33. For more fo-dicom sample code, visit https://github.com/fo-dicom/fo-dicom.

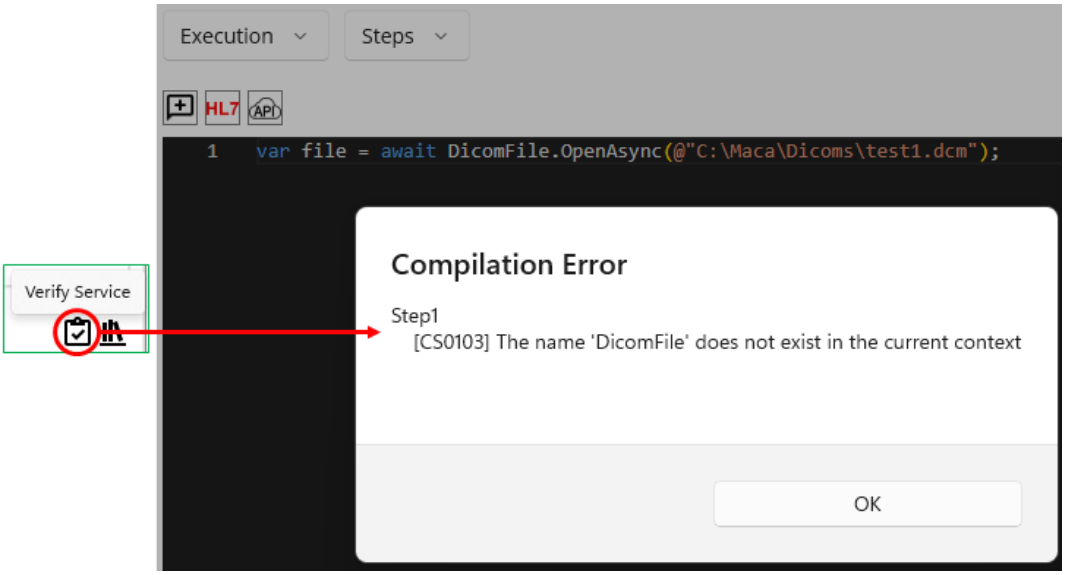


Figure 33.Missing Namespace Error

Since the DicomFile class belongs to FellowOakDicom, we can add "using FellowOakDicom;" in the Using tab, like in Figure 34.

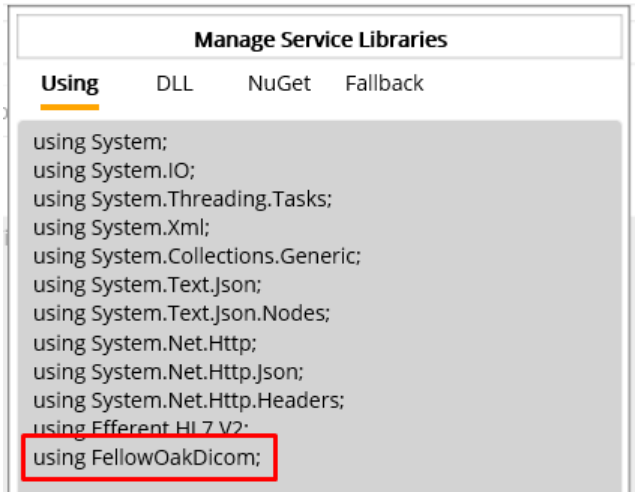


Figure 34.Add FellowOakDicom namespace

Now click the Verify button again to see if that fixes the issue.

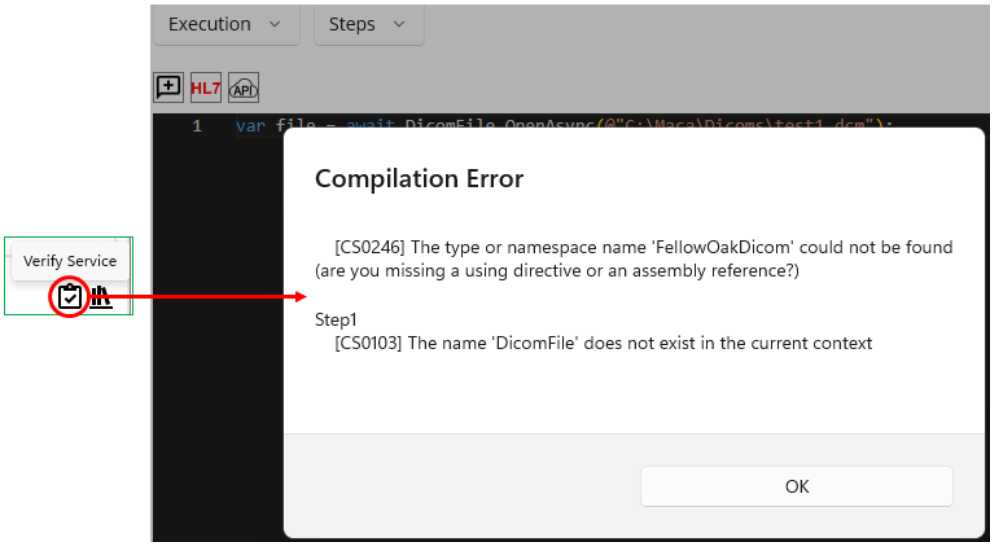


Figure 35.DLL Not Found Error

Although we added namespace to the Using tab, it still complains with an additional error message that the declared FellowOakDicom namespace is missing. As we downloaded the required DLL file but didn't add it to the DLL folder under the Instance, we now add the DLL.

DLL Tab: Add DLL to the Instance

Once you have the required DLL files, it is necessary to add them to the Instance DLL folder, which can be referenced by all services in the Instance. Another thing you need to remember before adding DLLs is that the version you selected has multiple target frameworks in most cases. See Figure 36.

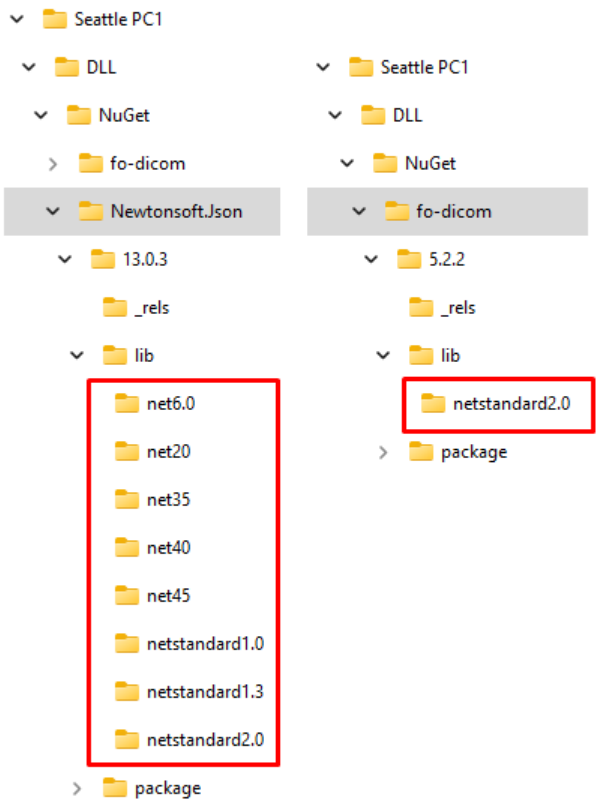


Figure 36.Target Framework comparison

As you see in Figure 36, there is a difference in supporting target frameworks. Newtonsoft.Json supports more target frameworks than fo-dicom, which supports only .NET Standard 2.0. Unlike the fo-dicom package, most NuGet packages will provide multiple target frameworks along with the latest .net version. To find out what .NET framework the package provides, let's take a look at the detail section below the version in Figure 37.

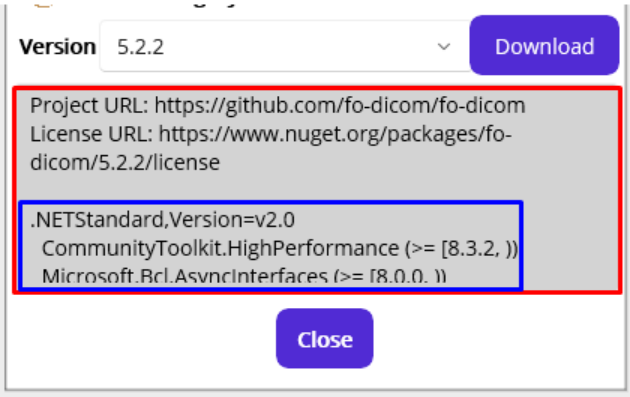


Figure 37.Package details with target frameworks.

Inside the blue box, there is helpful information about the target frameworks and related dependencies, i.e., the list of the DLL files that the package depends on, i.e., uses. You can scroll down the detail area to see more.

Note: Right below the framework name ".NETStandard,Version=v2.0" in Figure 37, there are a couple of the DLLs to which the package refers. This is another important part that you need to check when you use 3rd-party libraries. As the package "references" other DLLs, it is required to download or obtain them and put them in the DLL folder unless you already have them in the DLL folder or in the application. This will be covered later with an example.

With this information, add the DLL files to the DLL folder for the Instance. Click the DLL tab.

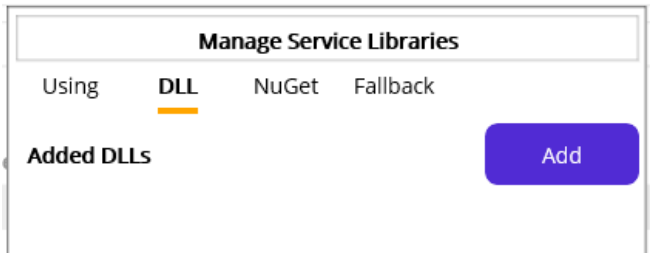


Figure 38.Empty 3rd-party DLLs

When you click the DLL tab, there is no added dll, as you see in Figure 38. Please click the Add button that pops up the File Picker. Using this File Picker, you can locate the dll file in the folder where you downloaded it from NuGet or in another location where you stored it.

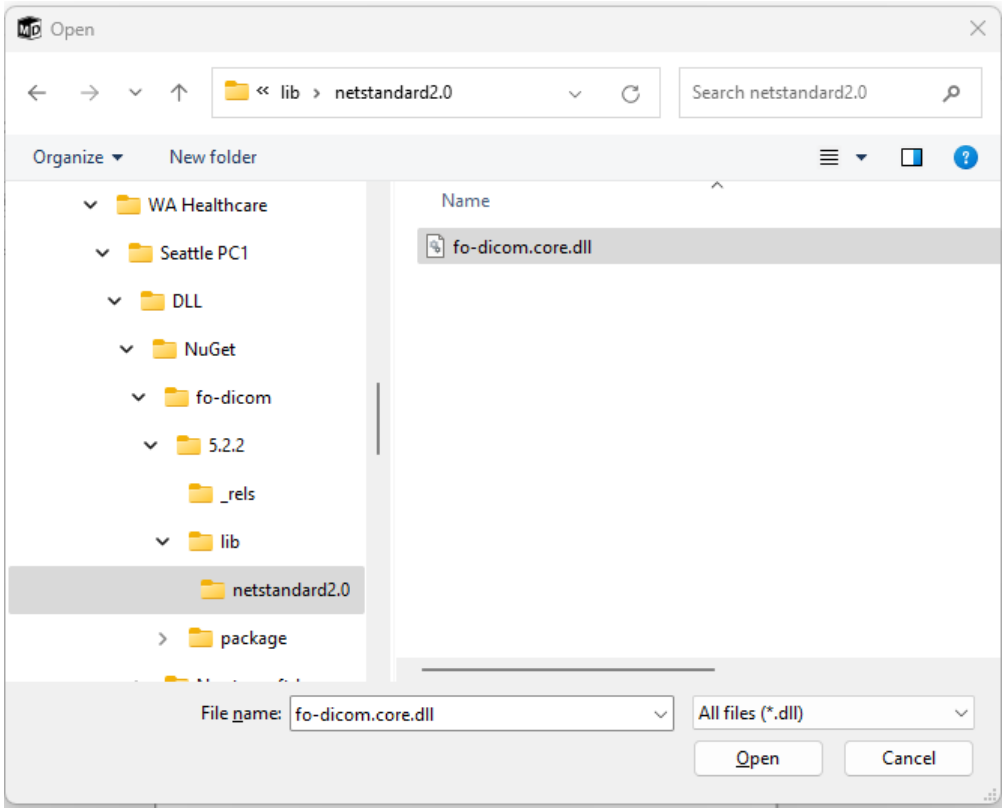


Figure 39.DLL File Picker

Select the “fo-dicom.core.dll” file and click the Open button. Then you will see the file is in the Added DLLs list and copied to the DLL folder under the Instance in Figure 40.

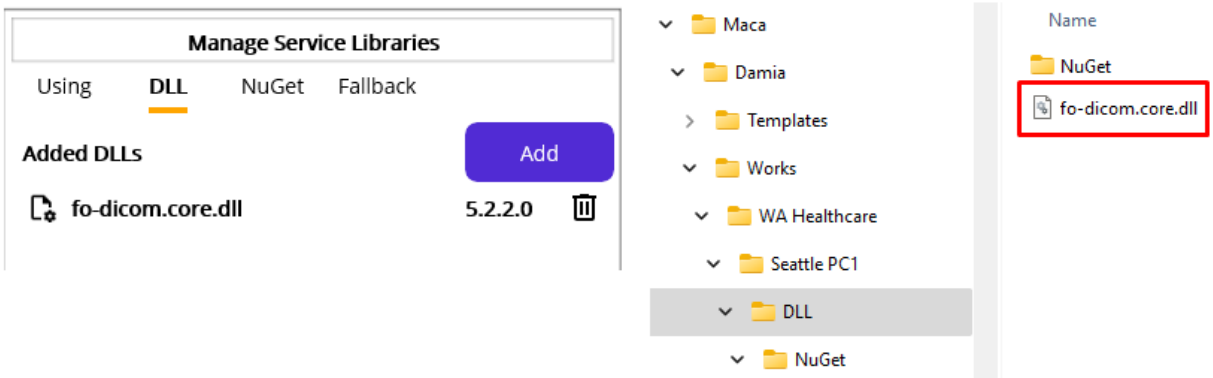


Figure 40.dll file added to the Instance

Now let’s try again the verification process. Click the Verify button.

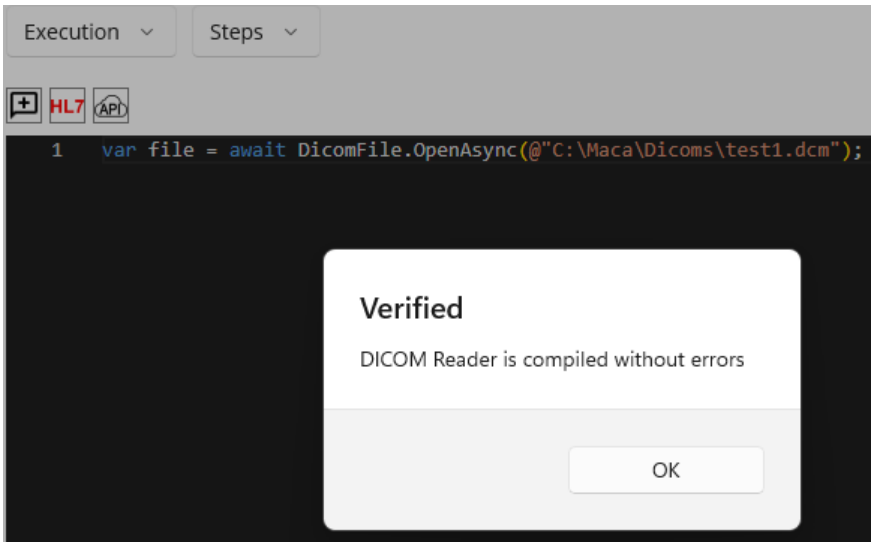


Figure 41.Verified Service

We saw that the code associated with fo-dicom.dll compiled successfully. Based on the above steps, we learned that the code verification procedure ensures the required DLL is in the Instance's DLL folder. That being said, we have to download the DLL, or obtain it from somewhere, and then copy it to the DLL folder all the time. However, you may forget these steps and just load the service on the Macaron server sometimes. In that case, the Macaron server will not complete the service deployment, and you may face unexpected results. To handle that exceptional case, especially a DLL-related situation, there is an option that you may put DLL fallback information in the service with Fallback. And we will cover how to load and deploy services onto the Macaron server later.

Fallback Tab: Auto-download specified DLLs to the server

As briefly mentioned, some cases, like when the service was loaded on the server without code verification or the dll was accidentally deleted, will produce unexpected results. To prevent such cases, you can provide the required dll information in the Fallback tab to save that information to the service itself. Click the Fallback tab button.

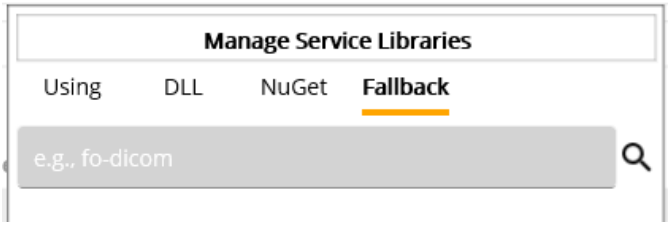


Figure 42.Fallback search

The UI looks similar to the NuGet tab since we have to search NuGet package ID here as well. In the ID field, type "fo-dicom" as we did in the NuGet tab and click the search button which is a magnifier.

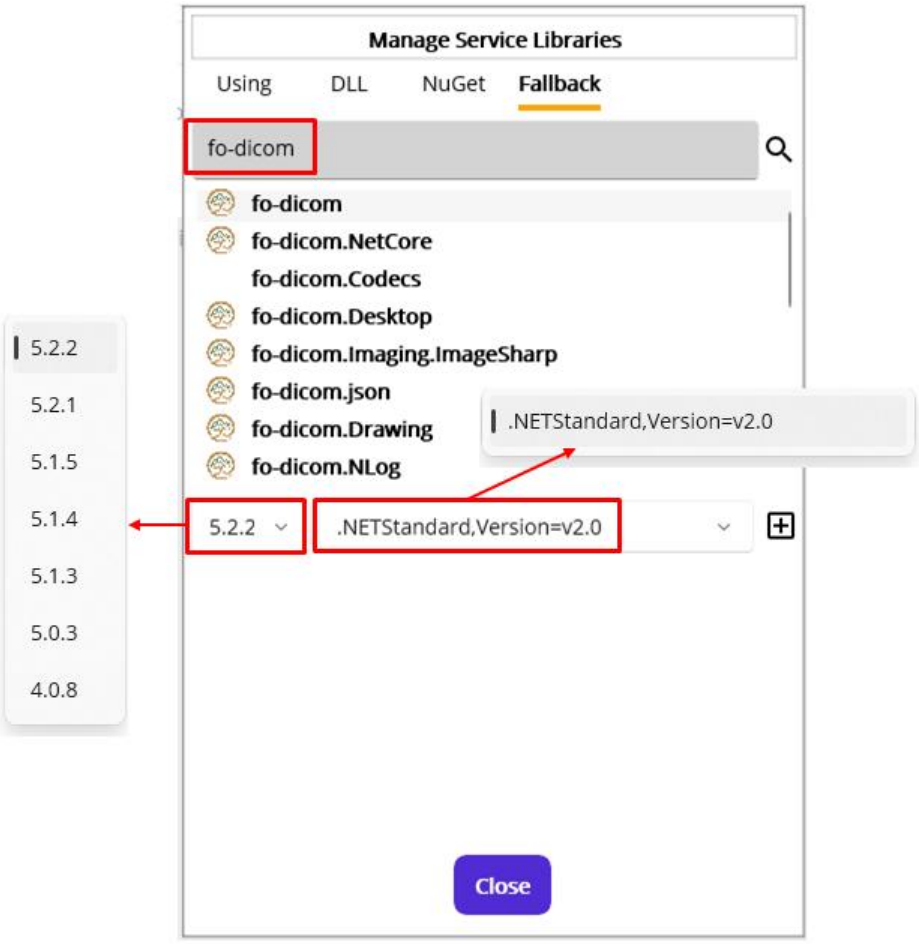


Figure 43.fo-dicom versions and target frameworks

Once you click fo-dicom that we want to add Fallback information to the service, its versions and supporting target frameworks will show up in Figure 43. As we saw before, fo-dicom supports only .NET Standard 2.0 and has only one in the target framework picker. But there are still available versions in the version picker on the left. Unlike what was shown in the NuGet tab, it doesn't show the details about the package. Please check that information in the NuGet tab. Since the selected version and target framework are what we used in the previous steps, keep this information and click Add Fallback button on the right.

Figure 44.Fallback value added to the service

With these steps, you can add or remove any package with a specific version and target framework to the service. And that information will be used on the Macaron server as a DLL file recovery. We will cover how this will work in the Management workstation section.

Note: The main difference between the NuGet and Fallback tabs is where the DLL file will be downloaded. The NuGet tab is designed to download the package to the local machine, i.e., your machine, to help with service development. On the other hand, the Fallback tab is designed to download the package to the server machine based on the specified NuGet information in the service. So, Fallback information will not affect any service development process.

We just went through all the tabs in the Manage Service Library popup, and there is one last thing that needs to be clarified. Among the tabs, the Using and Fallback tabs will affect each Macaron service. If you have two services, Service 1 and Service 2, for example, once you set that information to the service 1 and switch to the service 2 using project explorer (model explorer) on the left, you will see the values in Using and Fallback tabs are intact, i.e., not modified. On the other hand, the DLL tab shows the same information across the services. Once you change the information and switch to the other service, the changed information will be shown. Now close the popup and let's continue the code we have to write.

As we see the fo-dicom sample code now working, let's extend the code a little more, like in Figure 45.

Figure 45.Read DICOM file

The above code implements the procedure for finding and parsing dcm files. As an extra verification step, there is a txt file creation code at the last line. Before we send the parsed values to the target service that is not created yet, it is necessary to check the extraction code actually works with the dcm files. To access each tag, you can take advantage of the DICOM dictionary page (https://github.com/fo-dicom/fo-dicom/blob/development/FO-DICOM.Core/Dictionaries/DICOM%20Dictionary.xml) in the fo-dicom github. Then you can use the keyword, e.g., PatientID and PatientName, to get the value.

Note: The Development workstation doesn't provide a feature to run the service, i.e., start the service without the Macaron server. Although it provides code verification, running services is the area of the Management workstation. Once you verify the code, it is required to load the service on the server to see if the service actually works as you designed it. In the above case, we need to check whether the code that extracts values from DICOM file actually works or whether the service can be deployed on the server, not to mention the code we added in the Deploy view.

Before we run the service on Ron, close the Instance and take a short break.

Close Instance

In case you want to close the application and come back later, you can close Damia using the close button[20] [x] on the application's top-right corner. Or, you can close the loaded Instance using the Close Instance button on the Instance toolbar, which keeps Damia open, as shown in Figure 46.

[20] Damia doesn't provide close the app menu other than this button.

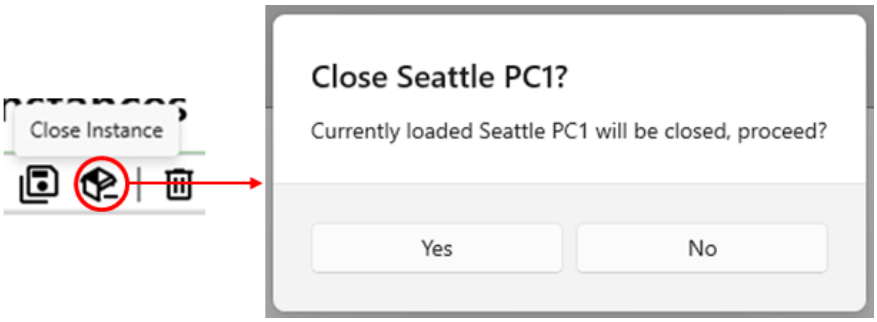


Figure 46.Close an Instance

Open Instance

Assume that you took a break and returned to this tutorial. Then you can open the Instance you closed to continue the work by using the Open Instance button on the Instance toolbar, like in Figure 47.

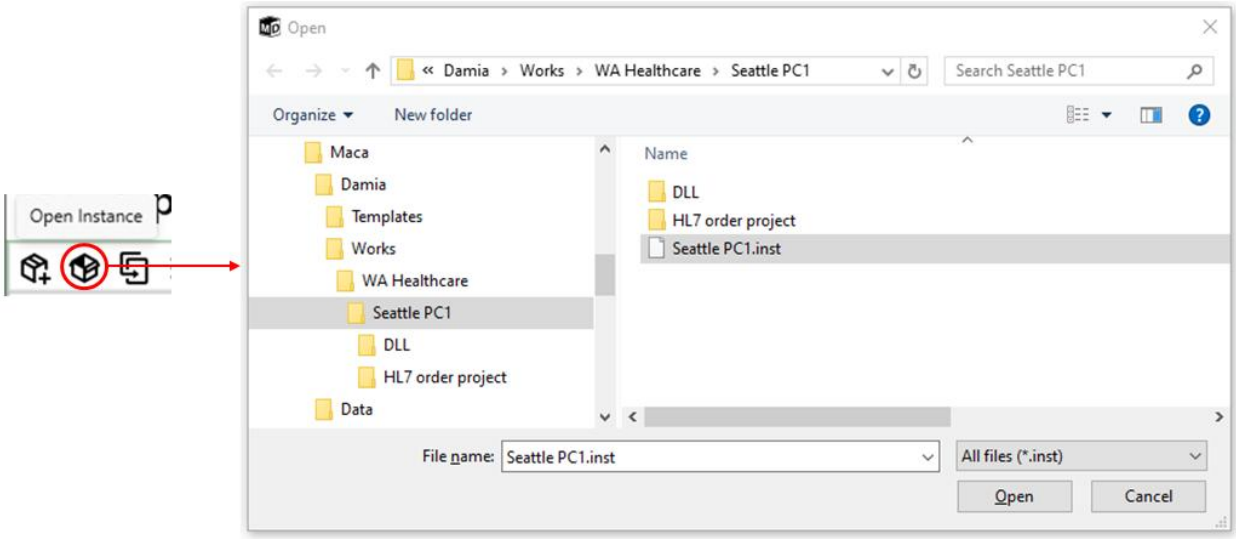


Figure 47.Open Instance

Alternatively, you can select an Instance from the recent works. Click the Recent Instances (☰) button on the Instance toolbar that will show Recent Instances popup, as shown in Figure 48. You can select any Instance from the list, and the location of the Instance will be shown right below the list.

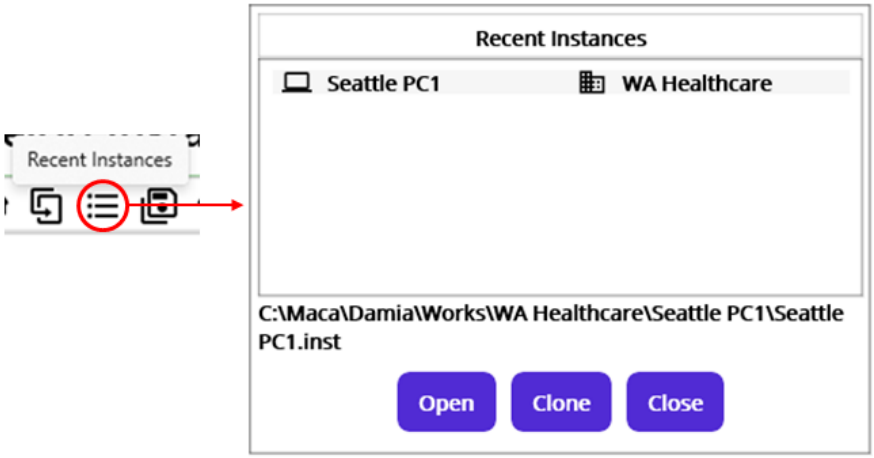


Figure 48.Open Instance from the Recent Instances popup

Select Seattle PC1 instance and click the Open button. For the Clone button, we will cover later.

Load Model to Server

As we finished the first code, it is time to check if the service can run on the Ron. In order to do that, it is required to load the service to the server first. Click the Load Model ( ) button from the Project toolbar.

Figure 49.Load Model requires model selection

Since we just opened the Instance and didn't select any model from the project browser on the left, clicking the Load Model button will show the popup with the "Please select a model to load" message. So, select DICOM Reader service from the project browser and click Load Model button again.

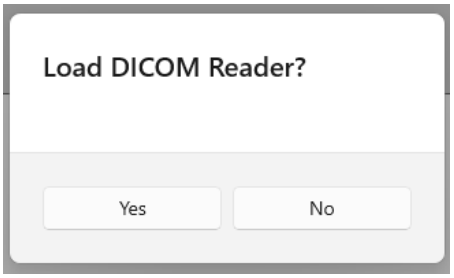


Figure 50.Load model confirmation popup

Once the popup asks you to load the selected DICOM Reader, click Yes button to proceed.

If you see below message that reminds you the local Macaron server is not available, you need to see if the server is down. If not, please start Ron immediately. Or restart Ron if necessary.

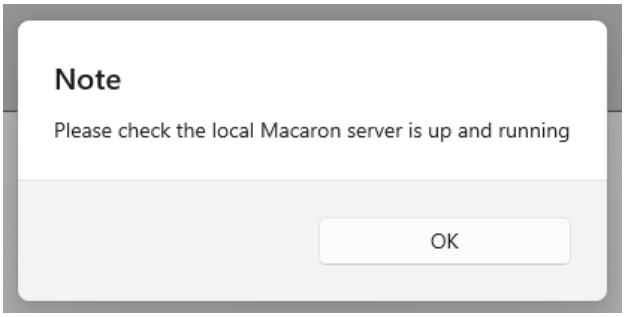


Figure 51.Ron must be started to load a model.

Once you start or restart Ron, come back to this step and click Load Model button again after selecting DICOM Reader service model.

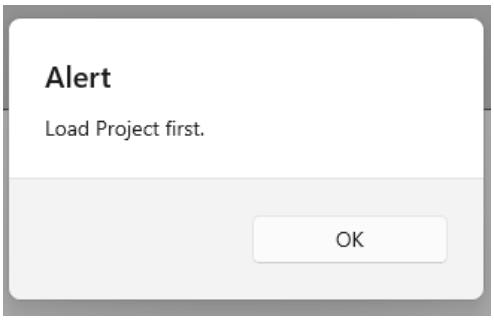


Figure 52.Alert if the service's parent doesn't exist on the server.

If you see this alert, as shown in Figure 52, it is time to recall the baseline of Macaron development. As we saw in the Create Instance section, the model needs to be created in a top-down way from Domain to Service order. This means that the child model will not exist without the parent model. In this case, the DICOM Reader belongs to the HL7 order project, and the HL7 order project must be loaded beforehand. However, once you select HL7 order project node from the project explorer and click the Load Model button, you will see a similar alert message like in Figure 53.

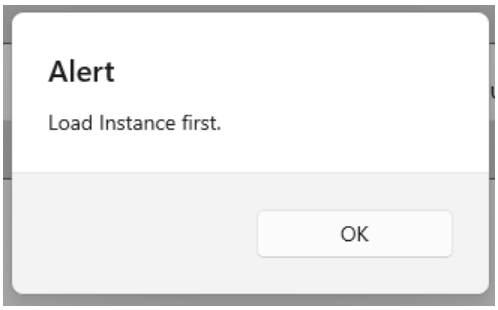


Figure 53.Alert if the project's parent model doesn't exist on the server.

In this way, we need to keep track of the hierarchy and check if the parent exists. But we haven't loaded any models yet and need to load a Domain model eventually. Select WA Healthcare node and click the Load Model button.

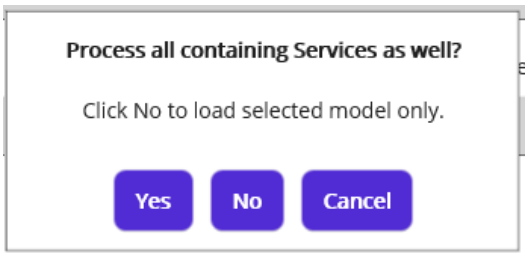


Figure 54.Group model load message.

As you see in figure 54, there are three buttons: Yes, No, and Cancel. Once you select any of the group models, this message popup will be shown. If you click the Yes button, all sub-models will be loaded as well. On the other hand, if you click the No button, only the selected group model will be loaded.

Note: Maca keeps track of user actions and saves them to the embedded database, SQLite. The data will be used to monitor user activity on the Management workstation. Any manual model reorganization, such as moving or deleting models, using file explorer may cause unexpected results or break the project's integrity as well as consistency. So, it is recommended to use Management workstation to load models onto Ron. We will go through this process shortly.

Click the Yes button.

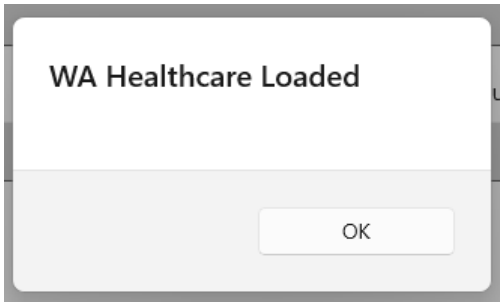


Figure 55.Domain load complete message.

Since we see the load is complete, let's move on to the Management workstation to check the models we loaded.

Management workstation

As we were at the Development workstation, let's switch to the Management workstation. When the Management workstation is loaded, you will see a similar look to the Development workstation. The left side is the Instance and Source managing area. And the right side is the selected model view area, as shown in Figure 56.

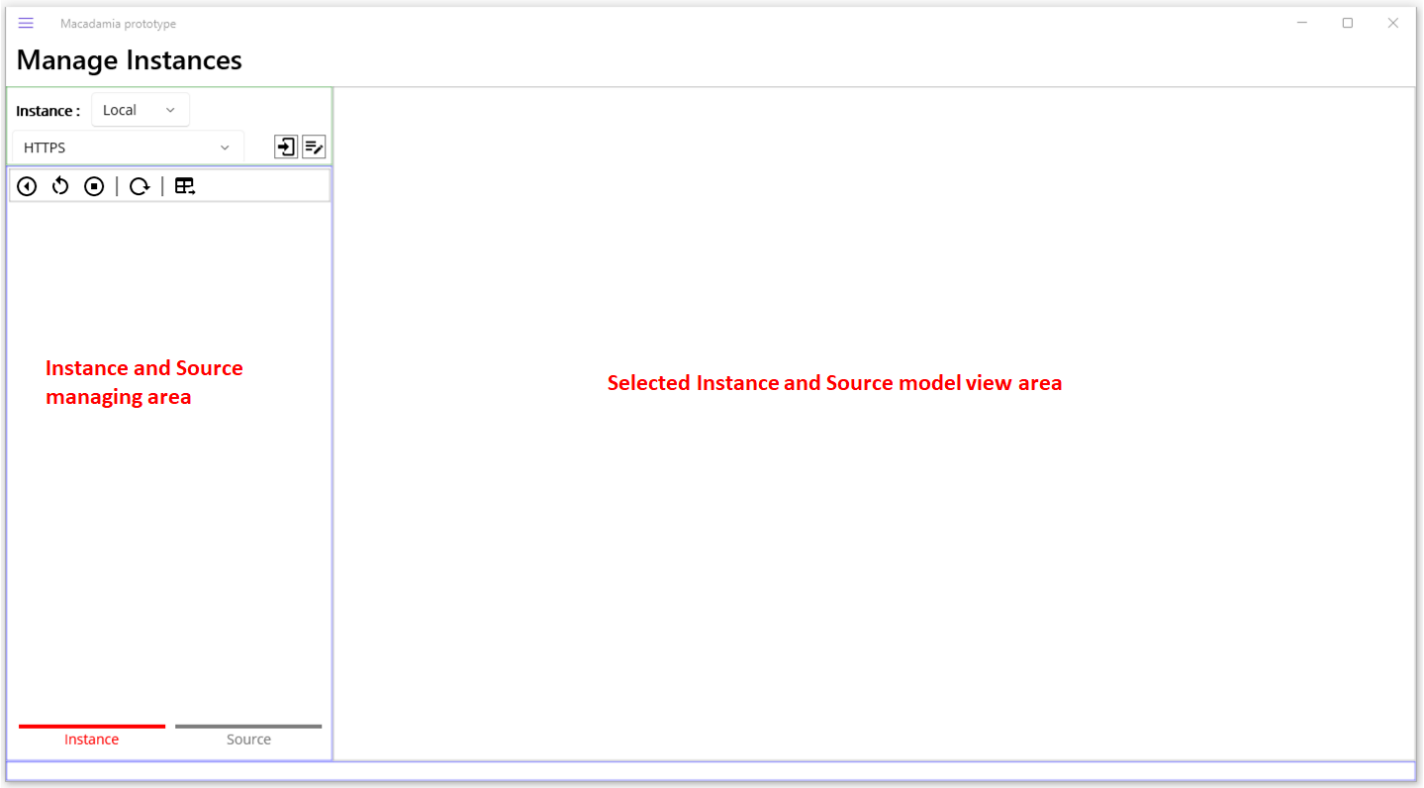


Figure 56.Management workstation

Once you open the Management workstation, you will see the Instance tab is selected. The Instance tab is designed to manage the deployed models, including running or stopped services on the Macaron server. On the other hand, the Source tab is designed to manage loaded models on the Macaron server.

Connect to the Macaron server

Default Users

The Macaron server provides user-based services for Damia. In order to use that, it is required to log in to the server, even if the server is installed on the local machine. When you install Ron for the first time, you can use the default user's credentials to log in to

Ron without creating a new user. See table 3 for the default users created after the Macaron installation. As you may see later, all these users will be retrieved in the Management Workstations. We will cover this in the Management workstation section.

Username	Open to Damia	Details
Admin	Yes	Full access to the Macaron server
Macaron	No	Takes care of the service start
LocalUser	No	Takes care of local model loading

Table 3.Default users

Among them, one thing to note is "Open to Damia." Unlike Admin, Macaron and LocalUser cannot be used as login credentials for the Macaron server because they are designed to be used for internal tracking. As we see later, they are visible only as a part of the retrieval result.

Note: The default user Admin's password is assigned with "Admin" for initial access. You can keep using this password, but it is recommended to change it unless you use Maca for testing purpose.

Server access panel

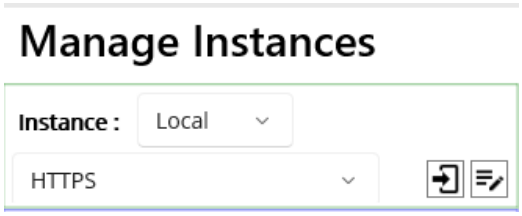


Figure 57.Macaron Server access panel

The server access panel, or simply a login panel, provides a way to choose the target machine's Ron, i.e., where the Instance is located. Please note that a target machine could be a local or remote machine. As you can see in Figure 57, the top row shows the target instance type picker. And the second row shows the target address picker along with the Login and Edit Address buttons.

Local Instance

By default, Local is selected for the Instance picker. As we loaded models on the local Ron, we can keep this selection. Right below it, there is HTTPS connection type and connection buttons next to it, as shown in Figure 58.

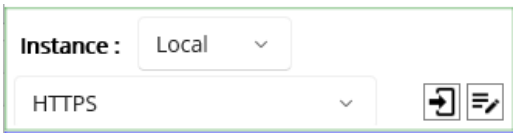


Figure 58.Local address connection pane

For now, we don't know what the actual value behind this selection is. To see the details, click the Edit Address ( ) button on the far right side, as shown in Figure 59.

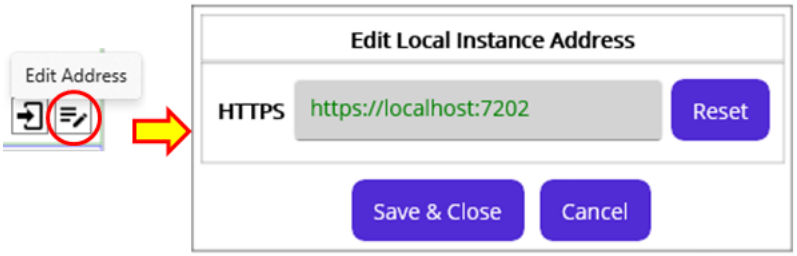


Figure 59.Edit Address popup

As you can see, there is a predefined address: HTTPS. Maca uses this address as a default value and will use it as restoring values. Once you change the address value, currently green colored, to a different address, you can restore it to its default address value by clicking the Reset button. For our case, the default https address, which is a localhost with default port 7202, will be used.

Alternatively, 127.0.0.1 instead of localhost will be accepted as well. Do not change any value, and click the Cancel button to close the popup. ***For prototype version, it is required to use fixed address https://localhost:7202 or https://127.0.0.1:7202 for connection***.

Note: There could be a situation where the address and port are already occupied by another application on the same machine. Macaron server uses gRPC protocol to communicate with Macadamia. Since gRPC provides port sharing, you can keep using this default address and port values unless you want to change them with other values.

Remote Instance

Note: ***Prototype version of Maca doesn't support the remote instance access and maintenance***. Although following demo insructs how to modify the address, this feature will be implemented in the later version. For now, Macadamia and Macaron installed on on the same machine will work.

Before we login to the local Ron, just take a quick look at how to access to a remote Ron. Choose Remote from the Instance type picker, as shown in Figure 60.

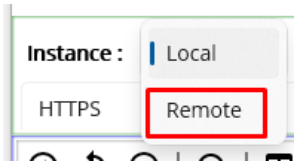


Figure 60.Switch to the Remote Instance

Once you switch to the Remote Instance, you will notice that the target address picker has no items to pick. Since we haven't added any remote Ron addresses, let's add the local Ron as a remote server for testing purpose. Because the connection is established with IP address and port, we assume that the local Ron can be a remote server as well. Click the Edit Address button.

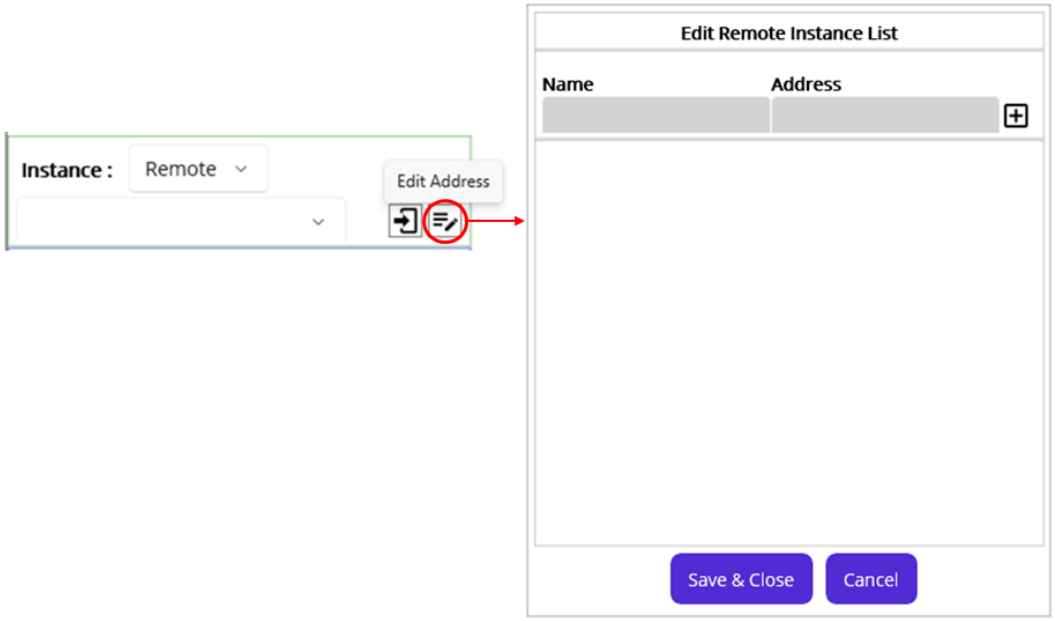


Figure 61.Edit Remote Instance popup

As you can see in Figure 61, a different popup, the Edit Remote Instance List popup is shown because we selected Remote type. Let's type Name and Address with the default local IP and port values. You can copy https address from the local instance editor shown in Figure 59. Then Click Add button on the right.

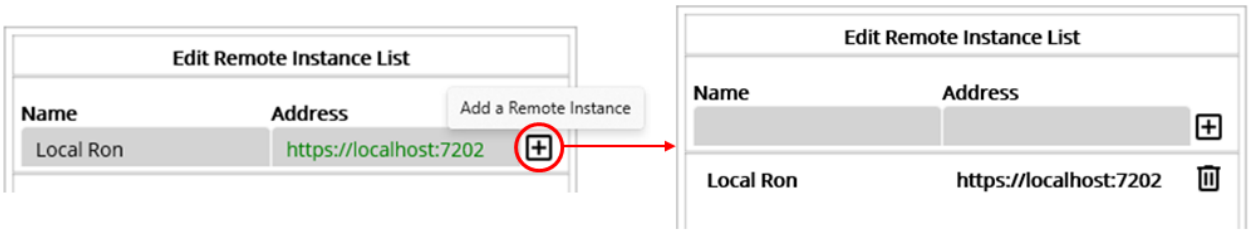


Figure 62.Add a remote Ron

Click the Save & Close button at the bottom. Then the new address will be shown in the remote address picker, as shown in Figure 63.

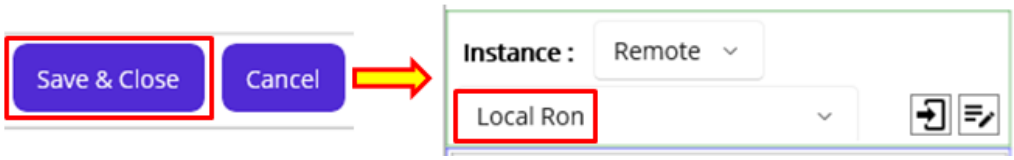


Figure 63.Added remote Instance

In the same way, we can add remote Instances that reside on the same network. Then we can manage models on a machine that doesn't have a Macadamia. Now, let's get back to the local access.

Log in to the Macaron

Once you click the login button, user login popup will show up. Please type default user credential, in our case "**Admin**" **for both username and password**, like shown in Figure 64. Then click the Sign In button.

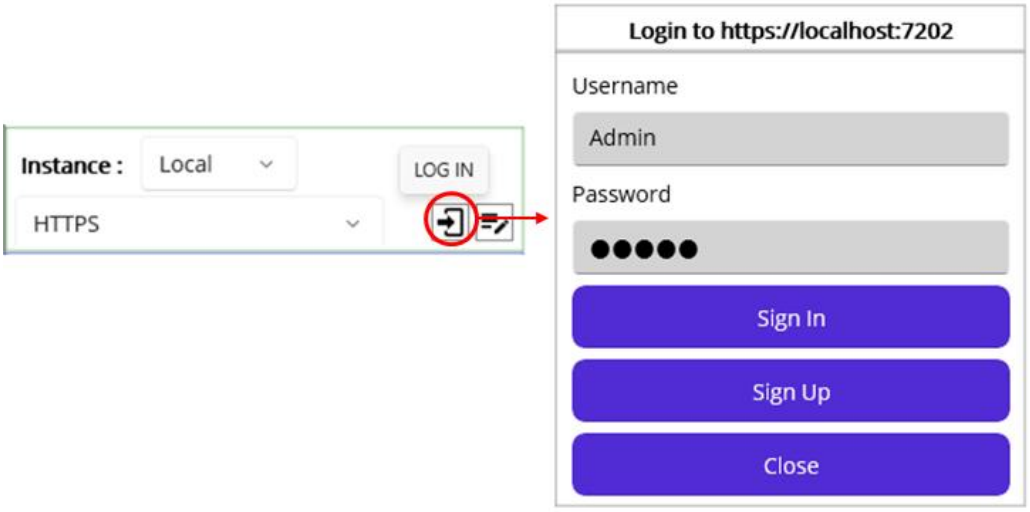


Figure 64.User login with default Admin credential

If the Macaron server is not started or the specified address value is not identifiable, you will see a connection failure message, like shown in Figure 65. In most cases, the Macaron server could be down or not started, so please check if the server is up first.

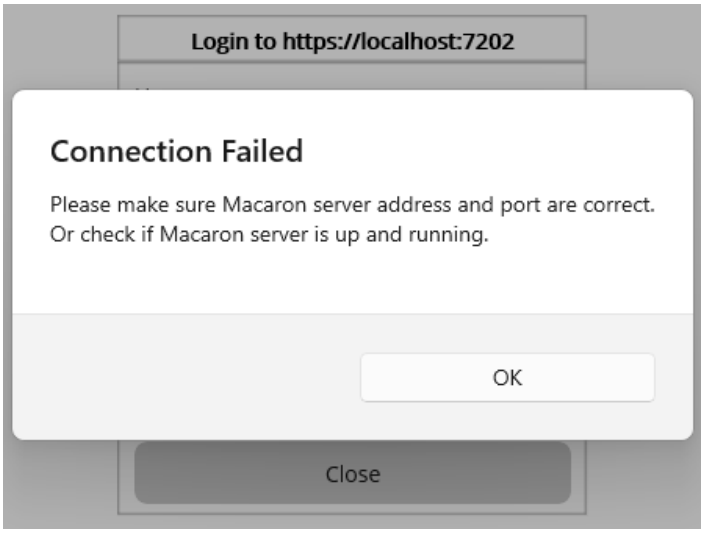


Figure 65.Connection failure message

Once you have successfully logged in, you will see that the username and the user edit button have appeared.

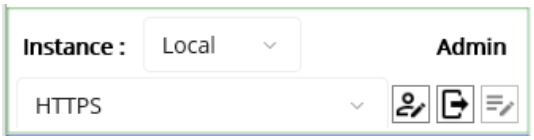


Figure 66.Server access panel after login

Additionally, you can edit basic user information using the Edit User ( ) button, as shown in Figure 67. Then you can change current user Admin's information here.

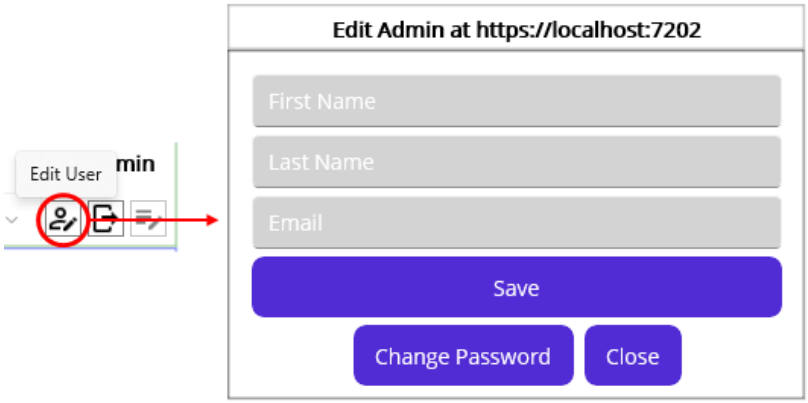


Figure 67.Edit User popup

Once you open the editor, "Admin" will be populated for both First Name and Last Name input fields. For demo purpose, that value was manually removed from the fields in Figure 67.

Note: The Instance model explorer is an area to show deployed models. Since this is the default selected tab when you open the Management workstation, any loaded model will be shown here after you log in. As we double-checked server availability when we tried to load the models onto the server, we assume that the server was not restarted or stopped after the models were loaded. So, there will be no deployed model in the Instance model explorer if you followed the above steps in this tutorial. In the event that you load model and restart the Macaron server, the Instance model explorer will contain deployed models. But for now, we assume that there is no deployed model after the login.

Instance Source location

For the prototype version, the default root directory for the Macaron server is following. When you click load ( ) button from the Development workstation, the selected models will be actually saved here.

C:\Maca\Ron\Instance

Source model explorer

As we just logged in, we can now browse services loaded on the local Macaron server. For now, there is no deployed model in the instance model explorer. So, click the Source tab to see the loaded models. Click the Source tab.

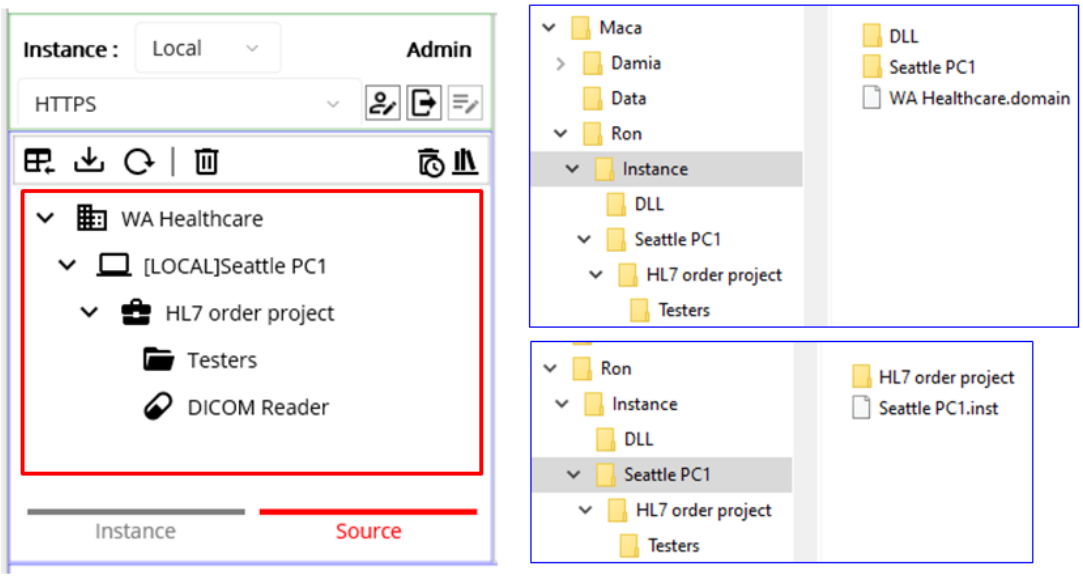


Figure 68.Loaded models (left) in the model explorer and the file structure (right) on the Macaron server.

As you can see in Figure 68, it looks the same as the one we saw in the Development workstation. The only difference is that they are on the server and will be deployed when Macaron starts up or you initiated. As the tab name Source describes, the model explorer depicts the physical file structure that you can see on the right side. Since Macaron contains only 1 instance at a time, there is no "WA Healthcare" folder like development. As a side note, DLL folder will be located right under the "Instance" folder.

For now, we didn't restart the server, and thus there is no deployed model. So, we can now deploy the models manually here using the Deploy button on the Source Toolbar. See below Table 4 for what each button does. We will cover each button's example later.

Icon	Name	Description
	Deploy	Deploys selected model and its sub-models(children)
	Load	Loads models to the server.
	Refresh	Refreshes to show up-to-date server side models in case models are loaded or unloaded.
	Delete	Deletes (Unloads) selected model from the server.
	Deactivate History	Shows the history of deleted models.
	DLL	Shows DLL files in the server.

Table 4.Source Model Toolbar

Source Model Viewer

Before we deploy the models, let's take a quick look at the source model viewer on the right side of the model explorer. Click WA Healthcare model.

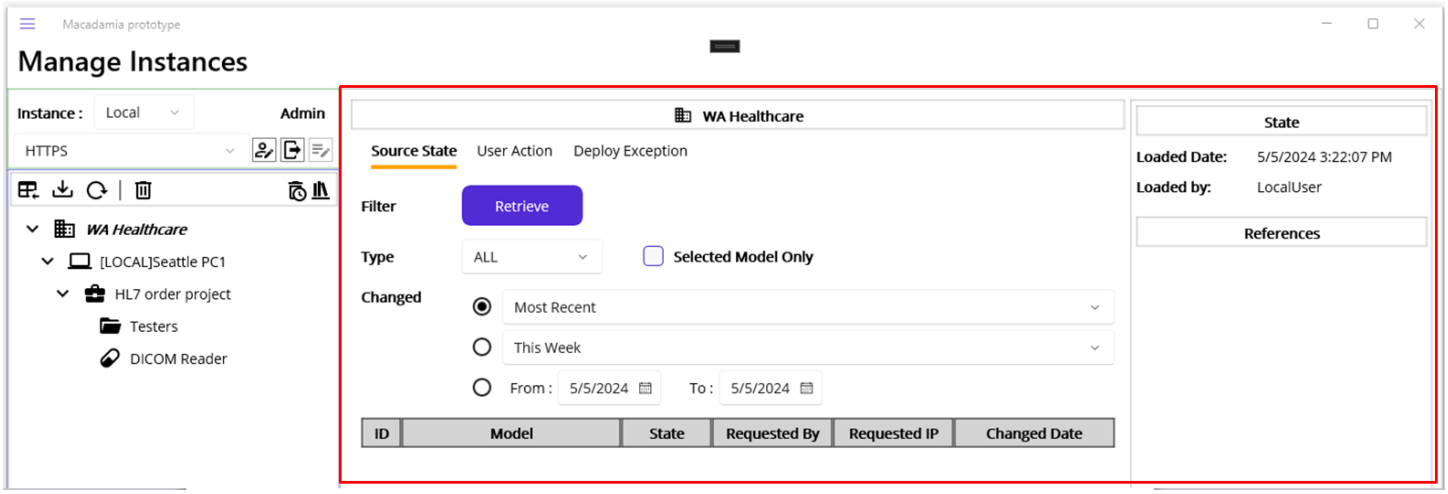


Figure 69.Source Model Viewer

Once you click any node of the model explorer, the detail information along with a retrievable view will be shown. Since we clicked WA Healthcare node, the view shows up the domain related data like Figure 69.

Source Model Detail

On the far right side, selected model's brief information will be shown like Figure 70. If selected model is a Service, more data would be displayed and we will cover that shortly.

Group model detail

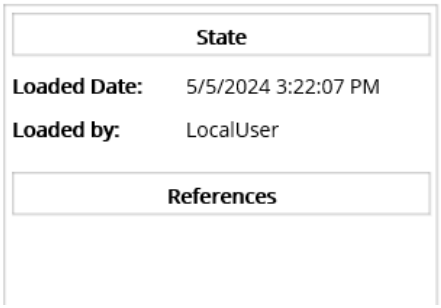


Figure 70.Group model's detail

As you loaded models from the Development Workstations, the file load information will be shown on top of the pane as Loaded Date and by. Among them, one thing to note is the LocalUser in Loaded by section. As we briefly went through the default users, LocalUser was used when loading models to the local Macaron server. And we can track who loaded the selected model by checking this value. There is another section, the References, to check and we will cover this later this tutorial.

Service Model detail

If you select DICOM Reader model from the explorer, the detail will look little differently, as shown in Figure 71.

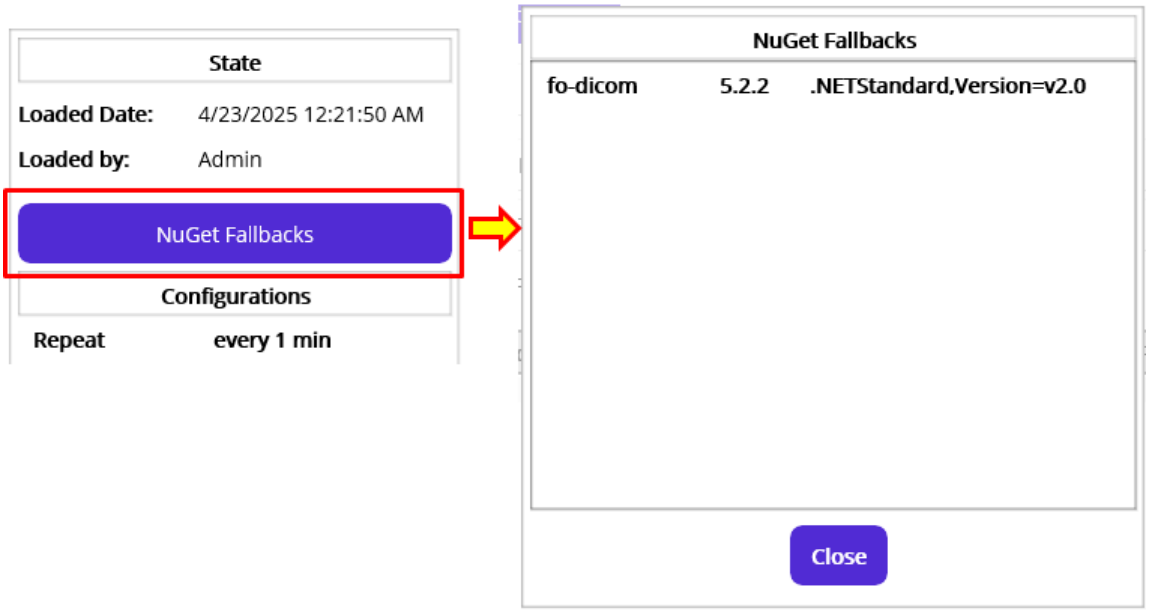


Figure 71.NuGet Fallbacks popup

Since you can define fallback values in a service editor from the Development workstation, the Management workstation also shows each service's fallback values in the detail view. Once you click the NuGet Fallbacks button, stored fallback values for the selected Macaron Service will be shown in the NuGet Fallbacks popup, as shown in Figure 71. We will revisit this popup later to show it as part of the exception handling.

Retrieve Source Model History

Using three center tabs, you can retrieve the selected model's history, such as when the selected model was loaded to the server.

Source State Retrieval

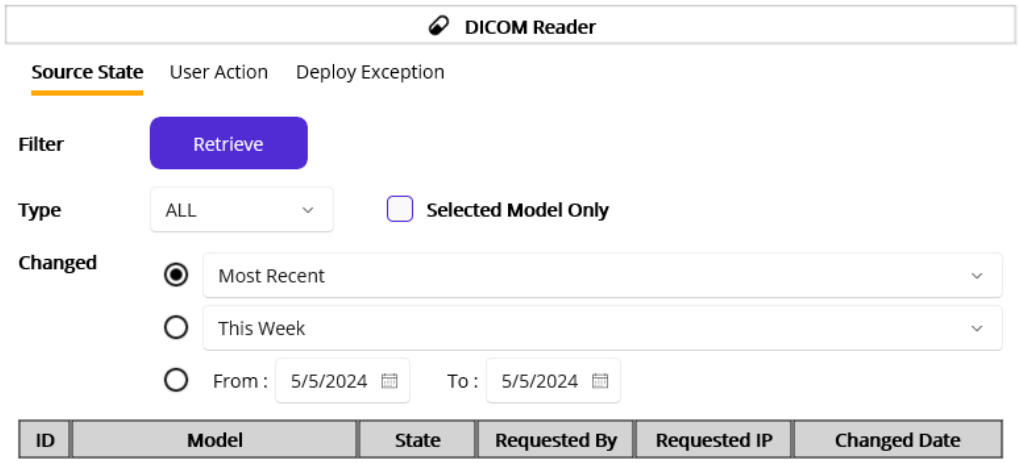


Figure 72.Source State Retrieval view

Once a user loads, deploys, or undeploys model(s), this section allows you to retrieve source file's state history based on the selected state types and changed (occurred) dates then shows the result in the bottom table. See table 5 for the filter options.

Filtering options	Details

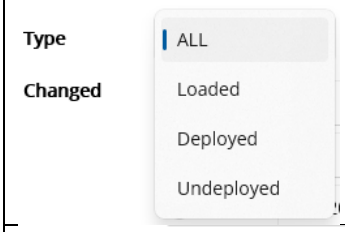
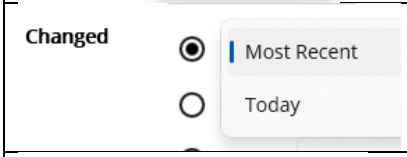
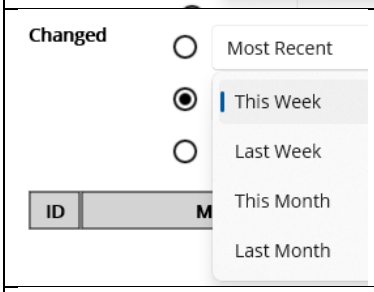
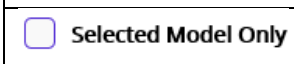
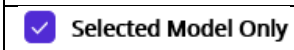
	**State Types** • **Loaded**: The model is loaded to the server after the Load command. • **Deployed**: The loaded model is deployed to the Instance after the Deploy command. • **Undeployed**: The deployed model was undeployed from the Instance after the Undeploy command
	Along with the selected type above, you can retrieve the most recent records or today's records.
	Along with the selected type above, you can retrieve a week's or a month's records.
	You can define specific time frame with this option.
	If a group model, e.g., Domain, is selected and has a child model, uncheck this retrieves all children's history records as well.
	If a group model, e.g., Domain, is selected, check this retrieves the selected model's history record only.

Table 5.Source State filtering options

Now let's retrieve some models' historical records with filtering options. Keep the current filtering options and click Retrieve button. Click WA Healthcare model from the explorer.

Retrieval example1: Group model and its children's state history

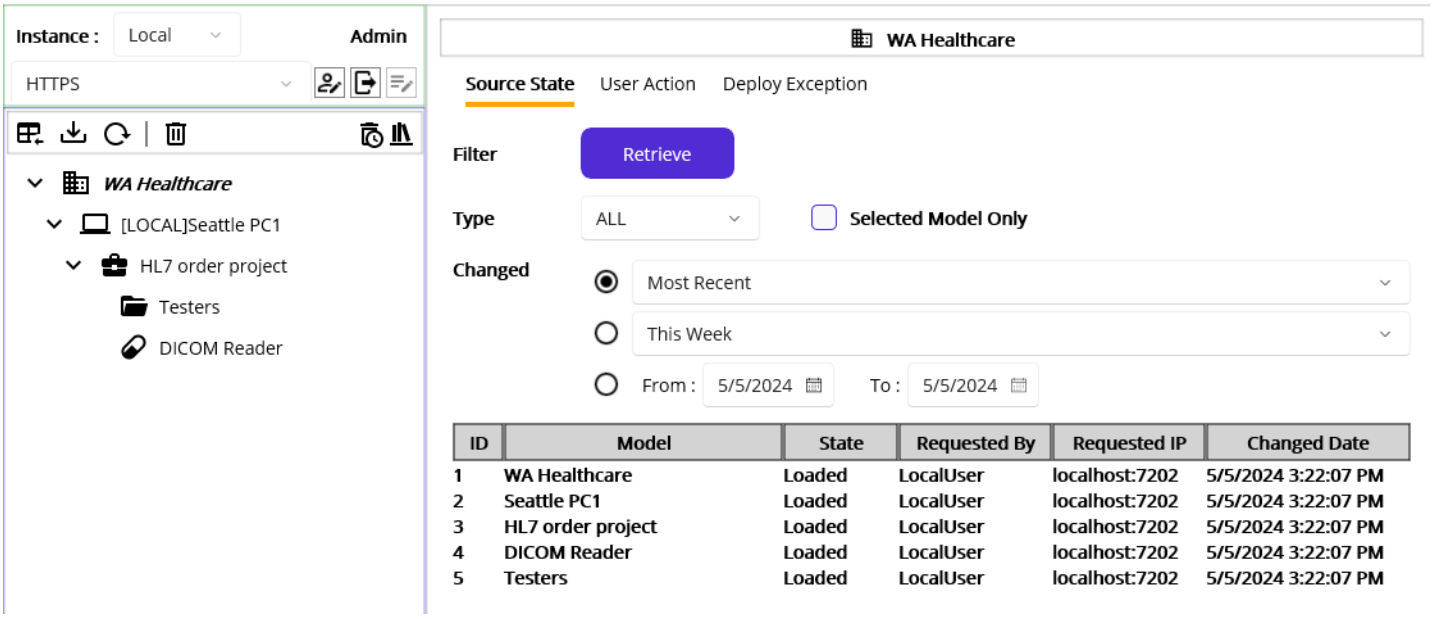


Figure 73.Domain model and its children's state records

Since we selected WA Healthcare that has children and didn't check "Selected Model Only" checkbox, all models' records were retrieved as shown in Figure 73. So, we can track the model's states based on the above filtering options. Likewise, if you select HL7 order project model and retrieve, all children's records will be retrieved, like in Figure 74.

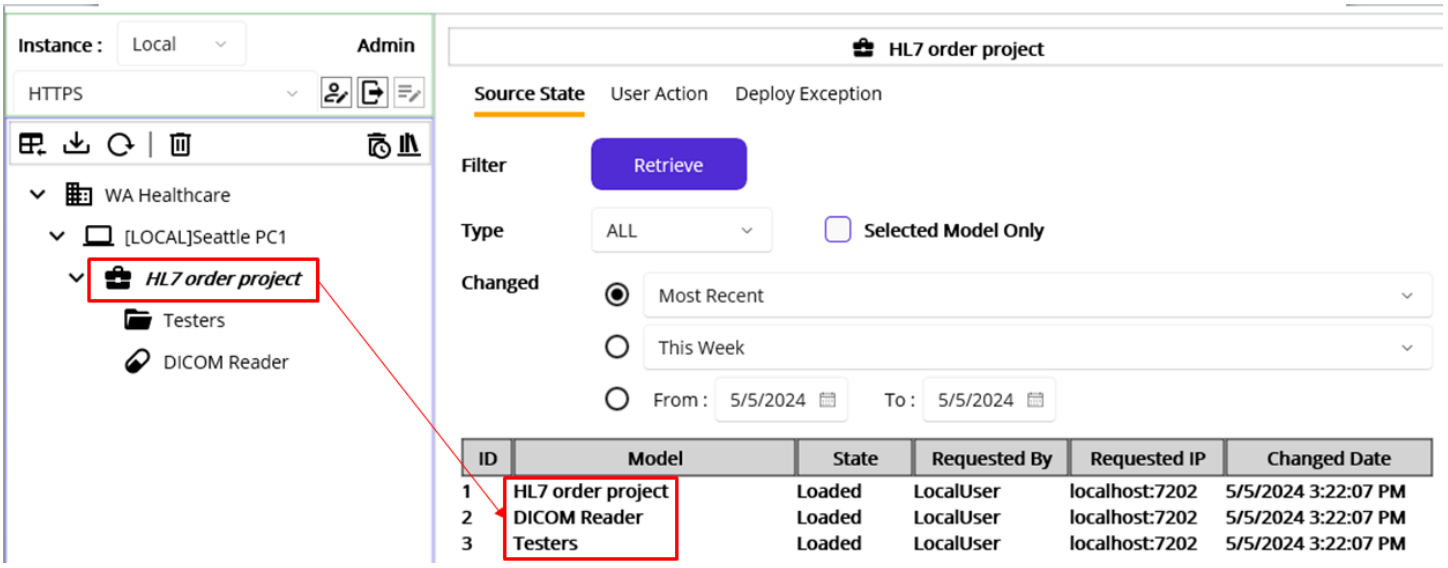


Figure 74.Project model and its children's state records

Retrieval example2: Group model's state history

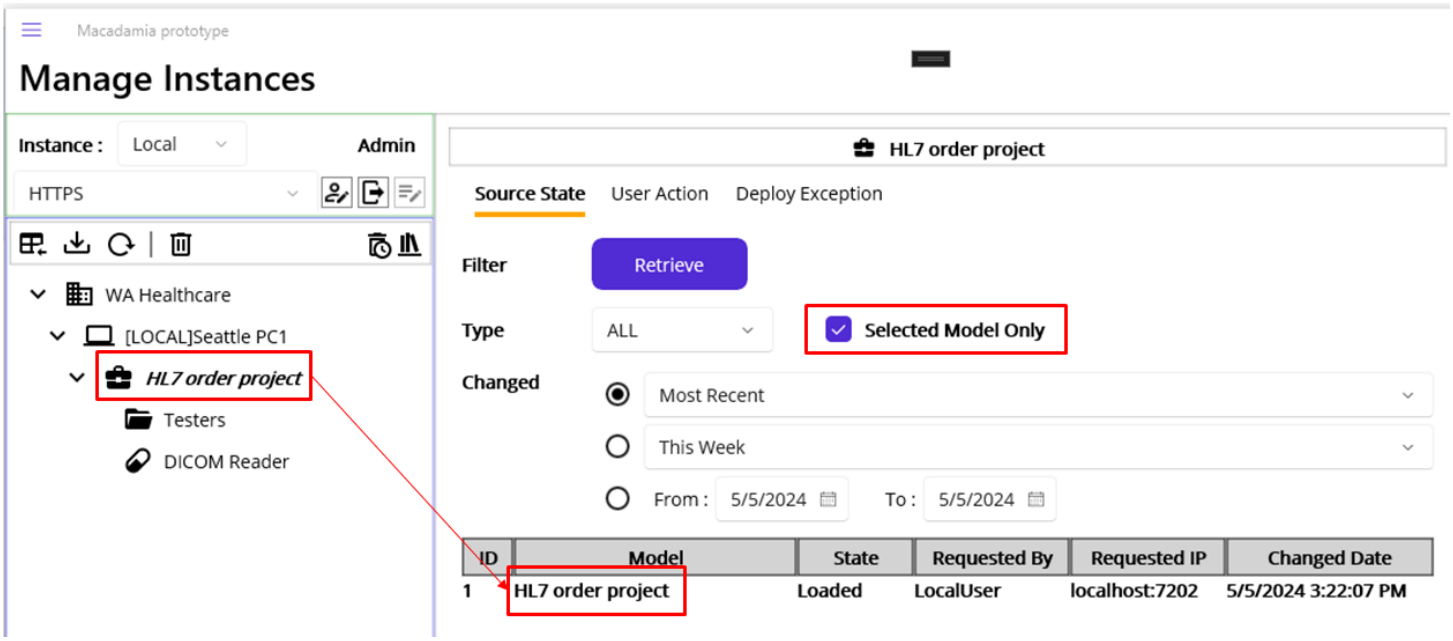


Figure 75.Selected group model records

Unlike Figure 74, if we check the "Selected Model Only" checkbox and retrieve, only the selected group model's record will be shown in Figure 75. If you want to track the selected model only, check the option. You can select other filtering options to retrieve more specific records.

User Action Retrieval

Click User Action tab. Similar to the Source State Retrieval, you can retrieve a history records for the user's actions: Load, Deploy, and Undeploy. As you may notice, the UI is almost the same as Source State Retrieval. The key difference here is that the result of the command will not be retrieved here; e.g., the result after the Deploy command will be retrieved in the Source State tab, not here. Since the Macaron server can be accessed by multiple users, tracking user action will be a key part of the source consistency as mentioned earlier. See table 6 for the filtering option details.

Filtering options	Details

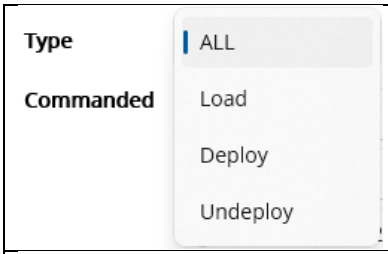
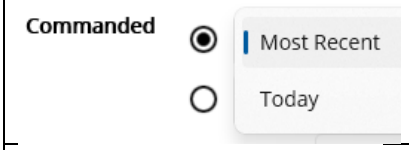
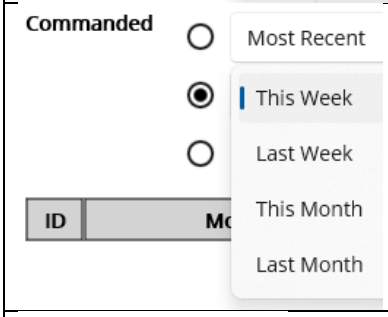
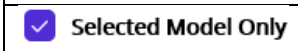
	**User Action (Command) Types** • **Load**: A user initiated Load command. • **Deploy**: A user initiated Deploy command. • **Undeploy**: A user initiated Undeploy command.
	Along with the selected type above, you can retrieve the most recent records or today's records.
	Along with the selected type above, you can retrieve a week's or a month's records.
	If a group model, e.g., Domain, is selected and has a child model, uncheck this retrieves all children's history records as well.
	If a group model, e.g., Domain, is selected, check this retrieves the selected model's history record only.

Table 6.User Action options

As we selected HL7 order project model from the model explorer, keep that and just click the Retrieve button.

Retrieval Example: Group model's command history

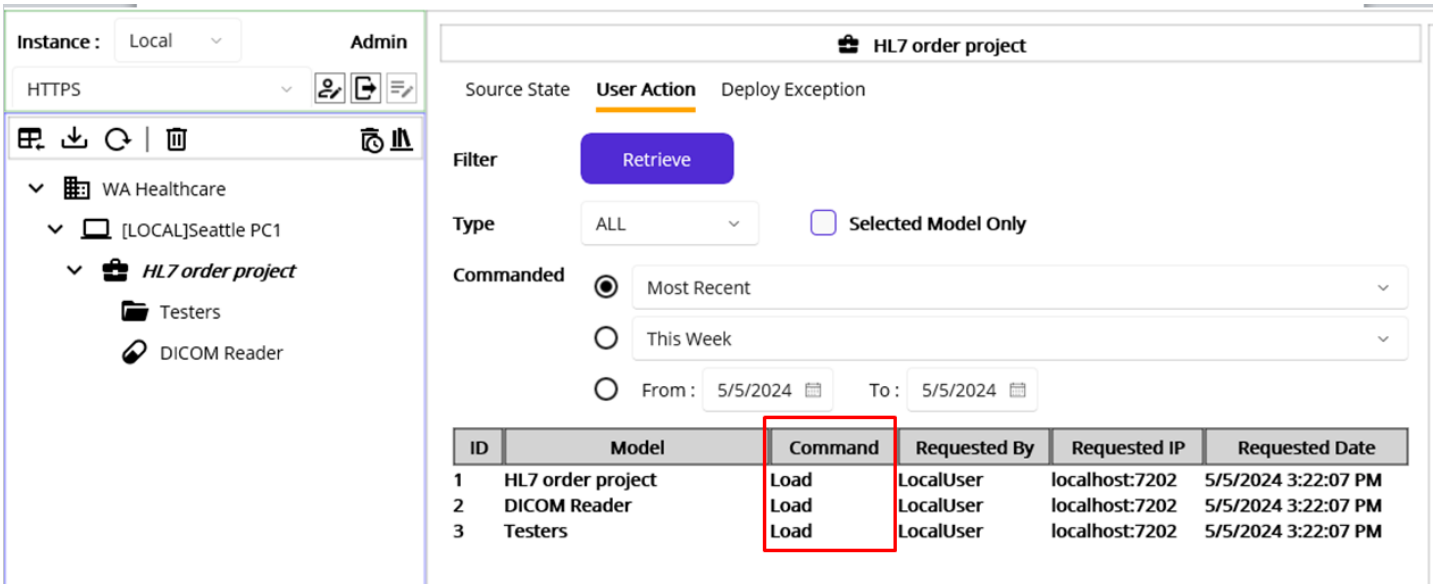


Figure 76.Group Model's command history

As you can see in Figure 76, Command column in the result table makes the User Action tab different. As all type is picked and Most Recent radio button is selected, we see the most recent command for selected models. Since the last action we've taken was load from the Development workstation, the result shows the Load command for the selected models. With this, you can focus on the

command itself here. Then you can check who and when that action was happened. All other filtering options are the same, and you can select them to see the different result.

Note: To make the difference more clear, we will revisit this section after the deployment sections.

Deploy Exception

Click Deploy Exception tab. This allows you to retrieve failed deployment history when you or Macaron deploys models. For now, we don't have any records since we didn't deploy model yet if we click the Retrieve button, like in Figure 77.

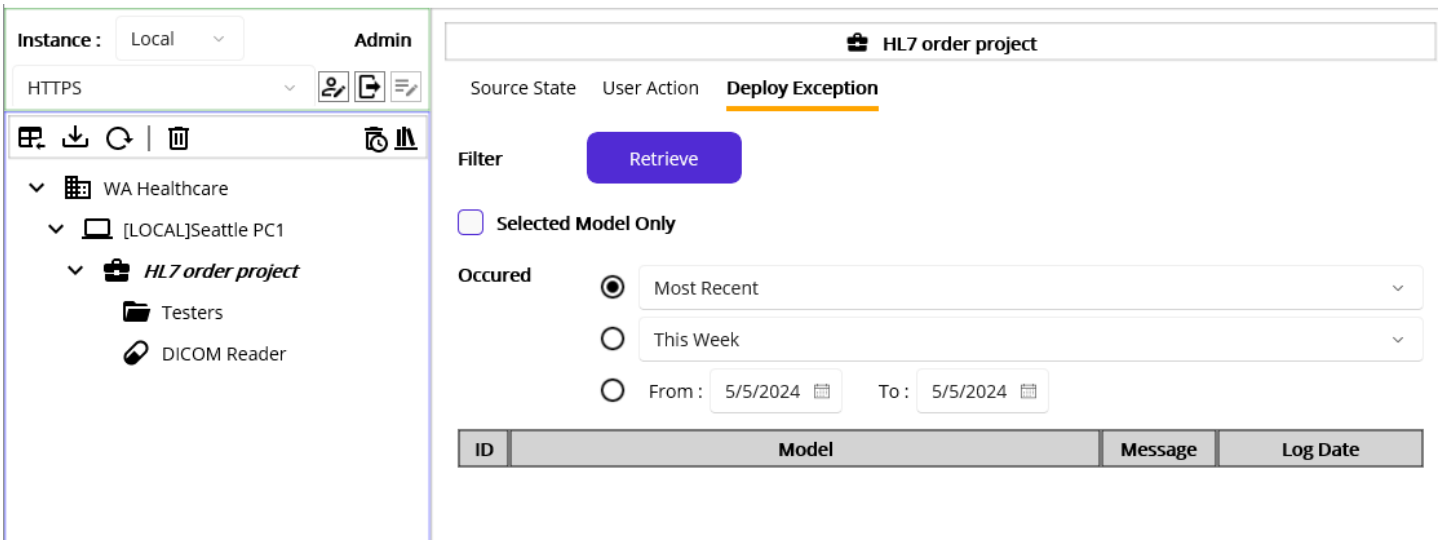


Figure 77.Deploy Exception options

Note: We will revisit this section and show more detail with some examples later.

Deploy Models

We loaded the models to the Macaron server and saw what the detail was. Now it's time to deploy them to see whether the services are working as designed.

Note: This could be an annoying reminder, but it is still noteworthy that you need to check the model's hierarchy in order to deploy the models. Before you select a model from the Source explorer and click the Deploy button, double-check if the selected model's parent model was deployed on the server. Otherwise, the alert message will pop up, as shown in Figure 78, which is the same process in the Development workstation.

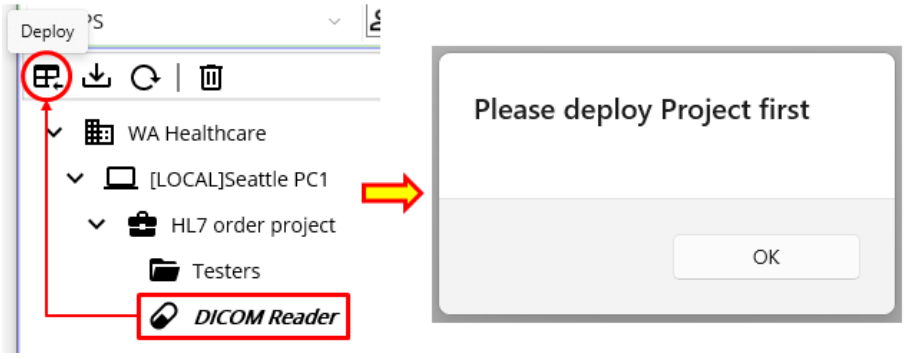


Figure 78. Alert when selected model's parent model wasn't deployed

Since this is the first time to deploy, select the Domain model, I.e., WA Healthcare, and click the Deploy model button.

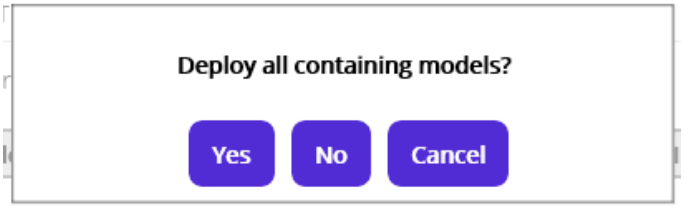


Figure 79.Deploy a group model

From the selection in figure 79, we can choose whether to deploy the selected model only or all sub-models as well, in case the model is group type. To see what will happen in the Instance model explorer if we choose the No button, click the No button.

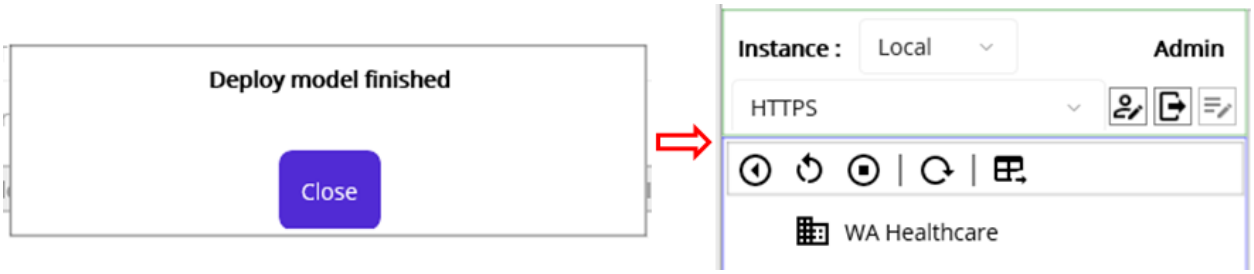


Figure 80.Deployed Domain model

Once you click the Instance tab after the finished message, you will notice that only WA Healthcare model is deployed, as we declined to deploy all sub-models.

Now, let's see the result of the first Deploy command in both Source State and User Action views. Click Source tab and go each view and click the Retrieve button. The result will look like below, as shown in Figures 81 and 82.

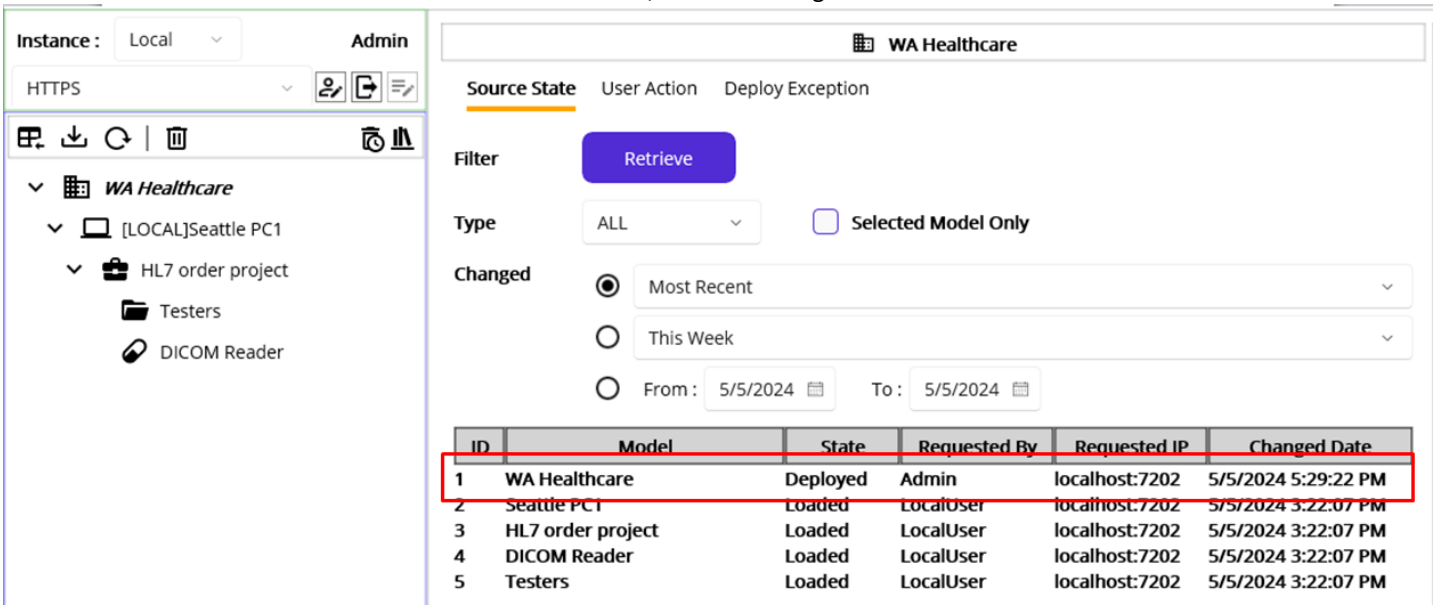


Figure 81.Source State records after the domain deployment

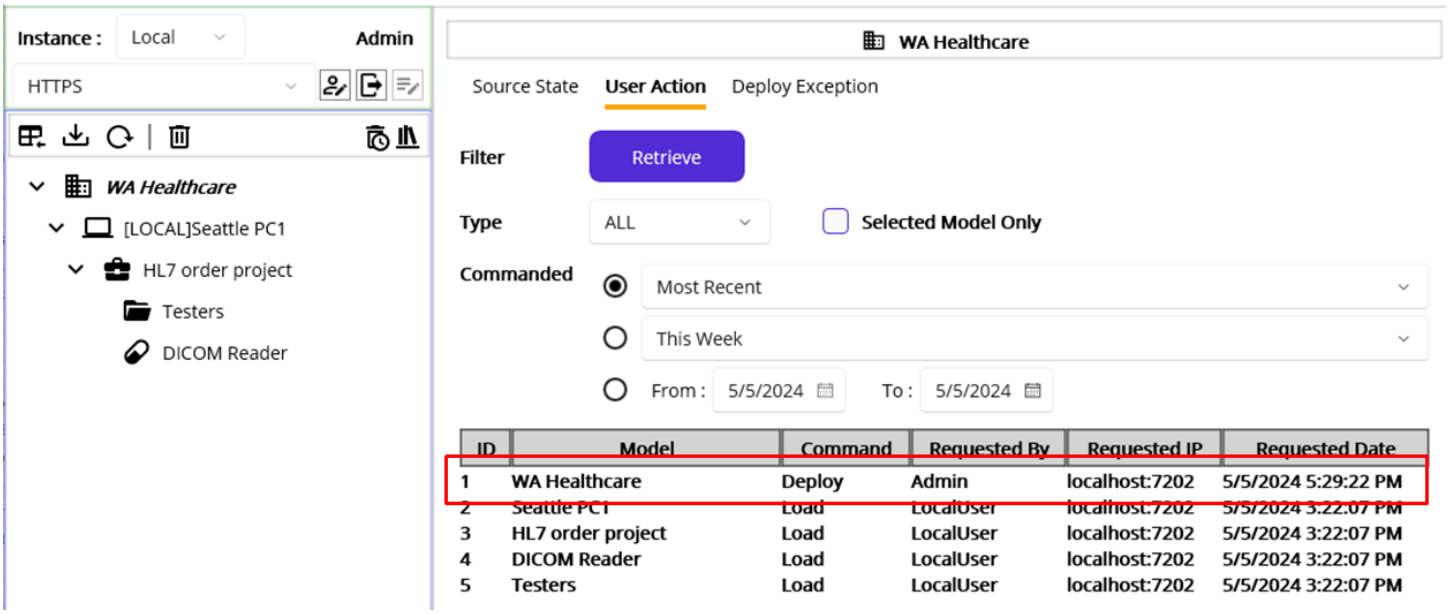


Figure 82.User Action records after the domain deployment

Note: There is no concept of start, stop, or restart for the group model itself. Once the group model has any Macaron Service, then you can start them as a group. For example, currently deployed WA Healthcare domain itself cannot be started, stopped, or restarted.

Now, we can deploy a domain as a whole when the popup asks. Select WA Healthcare if that is not selected and click the Deploy button. At this time, click the OK button, as seen in Figure79.

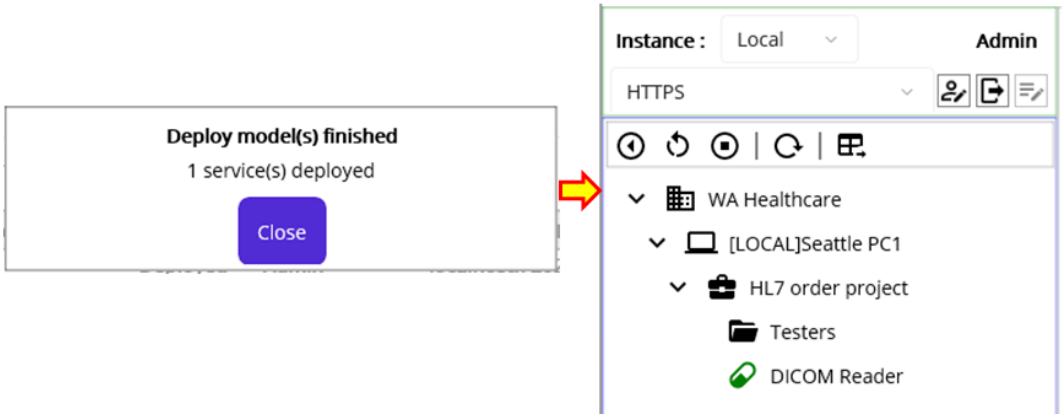


Figure 83.Deployed Domain with sub-models

Once you see the finished message popup, as shown in Figure 83, click Instance tab to see the result in the Instance model explorer.

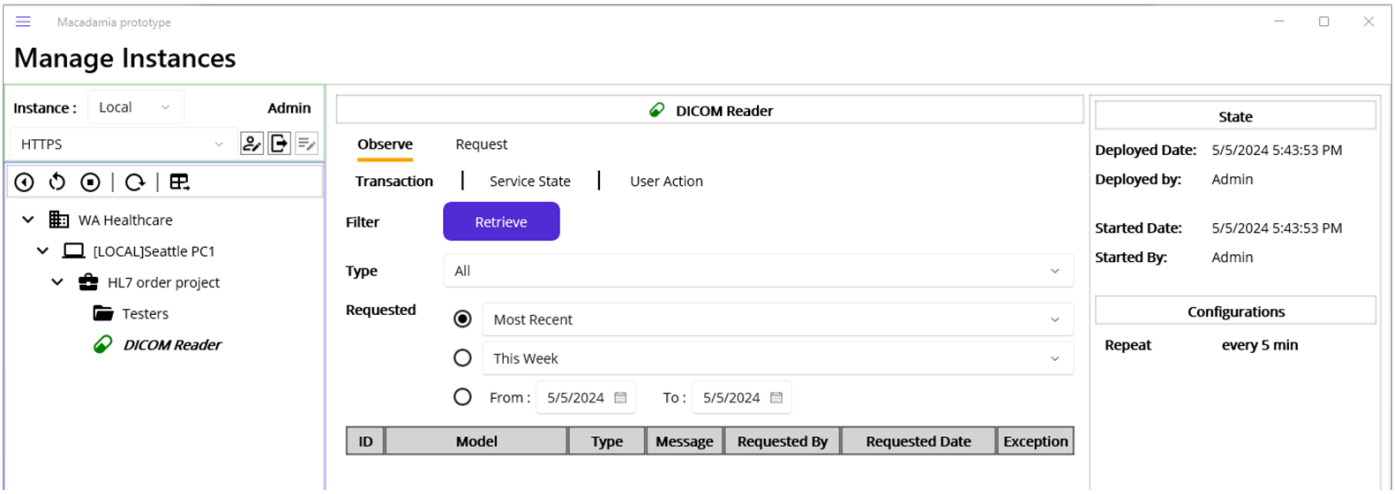


Figure 84.Deployed and running service

You can now click the DICOM Reader model to see the detail view, as shown in Figure 84. As you already noticed, the layout of the view is almost the same as Source tab's one. Let's take a look at each view.

Instance Model Explorer

Following Table 7 shows each button in the Instance Model Toolbar.

Icon	Name	Description
	Start Service	Starts selected stopped Service. If a group model is selected, starts all containing Services.
	Restart Service	Restarts (Stops and Starts) selected Service. If a group model is selected, restarts all containing Services.
	Stop Service	Stops selected running Service. If a group model is selected, stops all containing Services.
	Refresh Services	Refreshes models to show up-to-date instance models.(*Not implemented*)
	Undeploy Service	Undeploys selected model. If a group model is selected, undeploy all containing models.

Table 7.Instance Model Toolbar

Instance Model Viewer

Before we dig into each viewer, let's go over the term "Instance". An Instance model, interchangeable with a deployed model, is an object instantiated from the source model, i.e., the source file. Since the instance model is loaded in memory, it's ready to serve. As you saw earlier, instance model can be divided into two types: Group and Service. The group model is for organizing conceptual models and contains the References that can be referenced by Macaron Services. We will go over the References later. On the other hand, the service model, which is the core part of Maca, is a background service that executes the code you defined. With this reason, Instance tab and Source tab were used on the bottom of the left in the Management workstation. By switching back and forth the tabs, you can monitor the states of the models.

Instance Model Detail

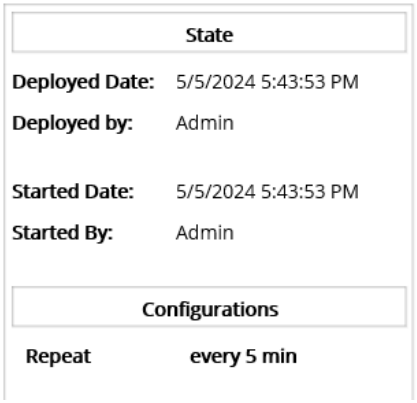


Figure 85.Instance Model Detail

Similar to the Source Model Viewer's detail view, it shows the State and Configurations information, as shown in Figure 85. You can check when this model deployed and started in State section. As shortly mentioned before, you can identify who deployed and started this service as well. Along with that, you can check the configured value for the selected service. As we set the DICOM Reader repeats every 5 minutes, it shows that information.

Observe Instance Model

You can observe deployed models' Transaction, Service State, and User Action history record here.

Transaction Retrieval

A transaction is a single job that was finished when the Macaron Service was triggered. For example, as we set DICOM Reader repeats every 5 minutes, one transaction will be created every 5 minutes. On the other hands, HL7Listener that we haven't touched yet will create a transaction when a request from an external app was received. We will cover that later as well. See below Table 8 for the filtering options.

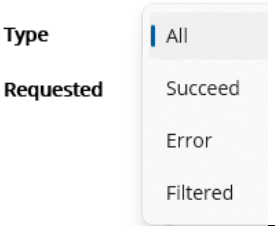
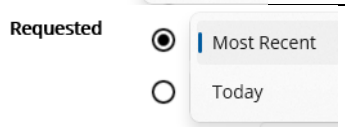
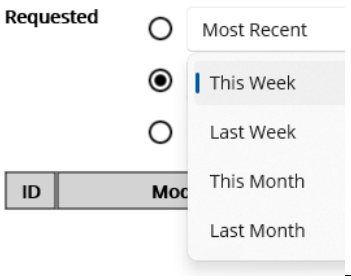
Filtering options	Details
	**Transaction Types** • **Succeed**: Successfully finished transaction. • **Error**: Transactions finished with errors. • **Filtered**: Request was filtered and not processed.
	Along with the selected type above, you can retrieve the most recent records or today's records.
	Along with the selected type above, you can retrieve a week's or a month's records.
	You can define specific time frame with this option.

Table 8.Transaction Retrieval options

Since we deployed and saw the DICOM Reader was successfully deployed, we can now retrieve the transaction records. Click the DICOM Reader service node from the instance model explorer. Since the initial view is Observe > Transaction, as shown in Figure 86, use the view as is and click the Retrieve button. You will see the most recent 1 record in the result table at the bottom. As you see, the transaction was finished with ERROR type. To view more about the detail, you can click the Message ( ) button and Exception ( ) button.

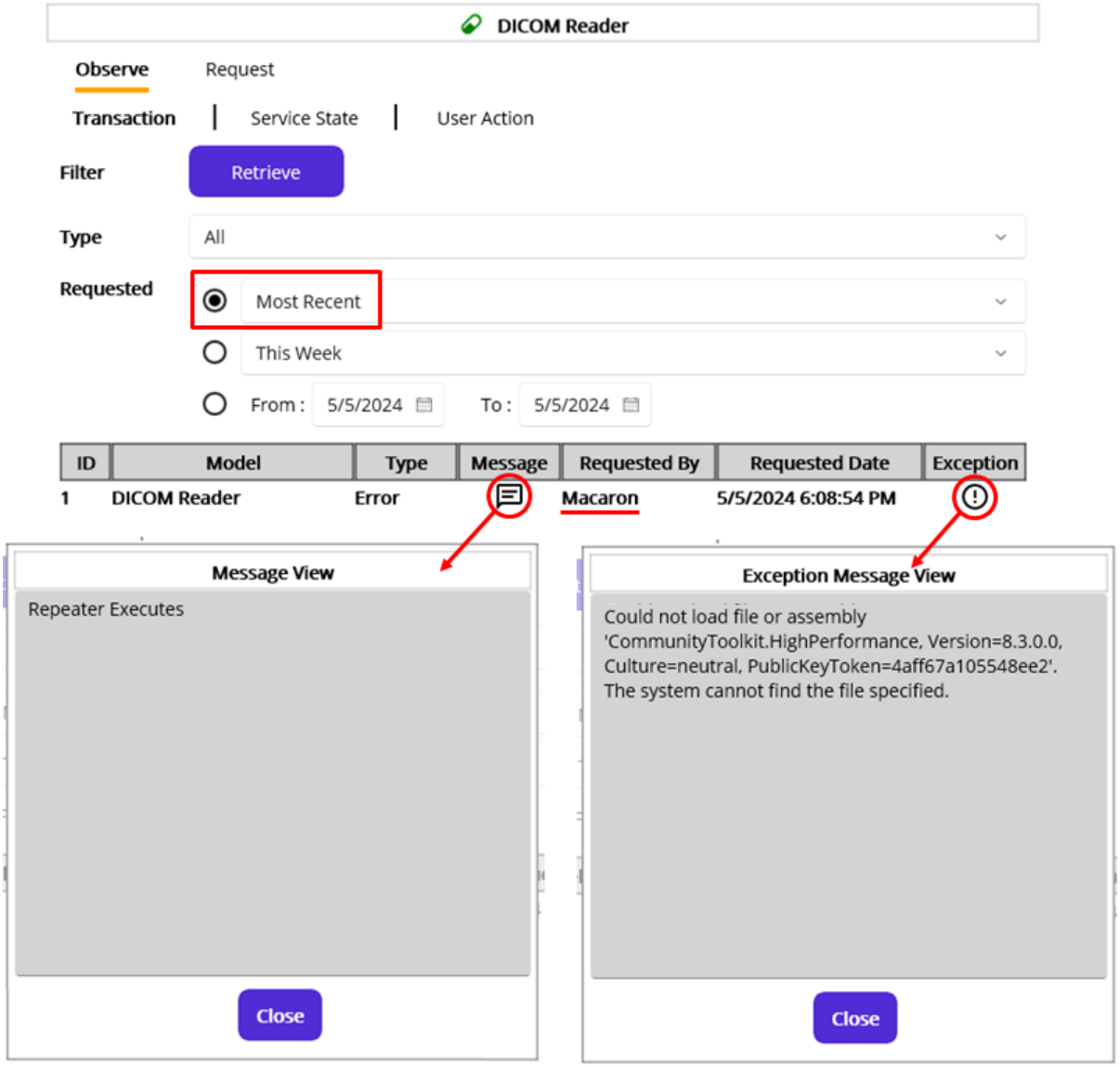


Figure 86.Retrieved most recent records

Note: As we briefly went over the default users earlier, you can see the Macaron user in the Requested By column in Figure 86. Currently we logged in as Admin, but the transaction wasn't created by Admin. The DICOM Reader was triggered by the Macaron user, which is a server, based on the 5 min interval that we configured. In case we send a request to any service, then the Requested By column will be populated with the Admin, or other logged in user who triggered that service. There is a Reprocess button, which is not implemented, at the bottom of the Message View popup in Figure 86. If a logged-in user clicks that button, Requested By column will be populated with the user name. However, the service polls every 5 minutes, or even shorter if we set 1 min, that feature may not be necessary and be removed in the next release. The button is for explanation purpose only.

Message View popup

Since the DICOM Reader is not triggered by any external request containing a Message data, every record in a result table will show the same “Repeater Executes” message when you click the message button. But the other services, such as HL7Listener or HL7Client, will show actual message in the received request. We will cover that later.

Exception Message View popup

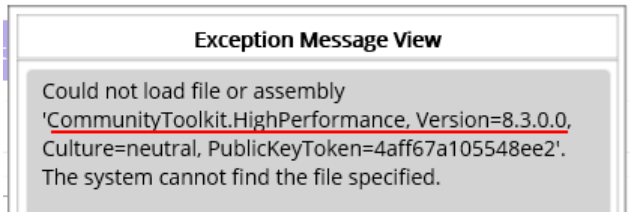


Figure 87.Missing DLL

When you click the Exception button in the result table at the bottom, any exception that occurred during the process will be shown in the Exception Message View popup, as shown in Figure 87. Although we verified the code itself using Verify ( ) button in the service editor, still there are some conditions that we may not catch at development time, such as the requested address is invalid or even down in case of web service handling. Like this exception, which is a missing required DLL file exception, when you use 3rd party DLL, this kind of exception may occur as well. We use fo-dicom to parse DICOM files and this internally requires another DLL named “CommunityToolkit.HighPerformance.” As briefly mentioned in **Figure 37** along with **Note** section, fo-dicom requires “**CommunityToolkit.HighPerformance**” DLL to work with DICOM files. Since we don’t have that, we need to obtain that from web or using NuGet tab in the Service Editor.

Note: This missing DLL exception is somewhat important point to remember in developing and deploying services to the Macaron server. Since we may put as many 3rd party DLLs as possible in the local Damia and Ron, we could easily fall into the unexpected error in the Macaron side while we deploying the services. Since you will handle various devices, you may forget to check the required DLL files or assume that they are already there. That’s the reason why each Instance has “DLL” folder and keep all required DLL in it, as shown in Figure 30 and 36, not to mention the Fallbacks section in each Macaron Service. Likewise, Macaron server will have the default “DLL” folder to keep all the required DLL files, as shown in Figure 88.

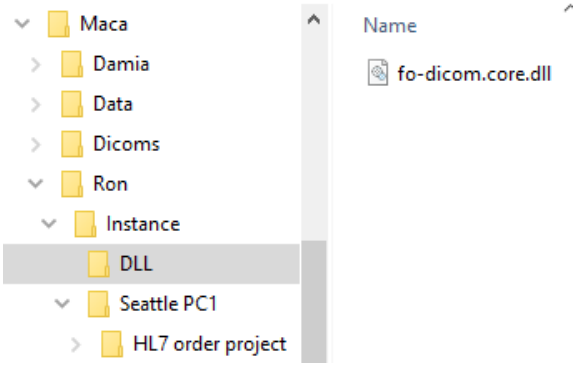


Figure 88.Macaron server's DLL folder

One thing you may notice in Figure 88 is the Dicoms folder. As we set the code in the deploy code editor, which creates a Dicoms folder if it does not exist, that code was executed when we deployed the service. See the Figure 24 for the code details. As an extra step for the test, put obtained sample DICOM file to the Dicoms folder. Again, this tutorial doesn’t provide a DICOM. Since this uses a sample file found in NEMA site, the result data would be different from your test case with other DICOM files.

Instead of browsing the file system to look at the DLL files on the server, you can check what kind of DLL your Macaron server has using DLL ( ) button in the Source Model Toolbar to show Loaded DLL popup, as shown in Figure 89.

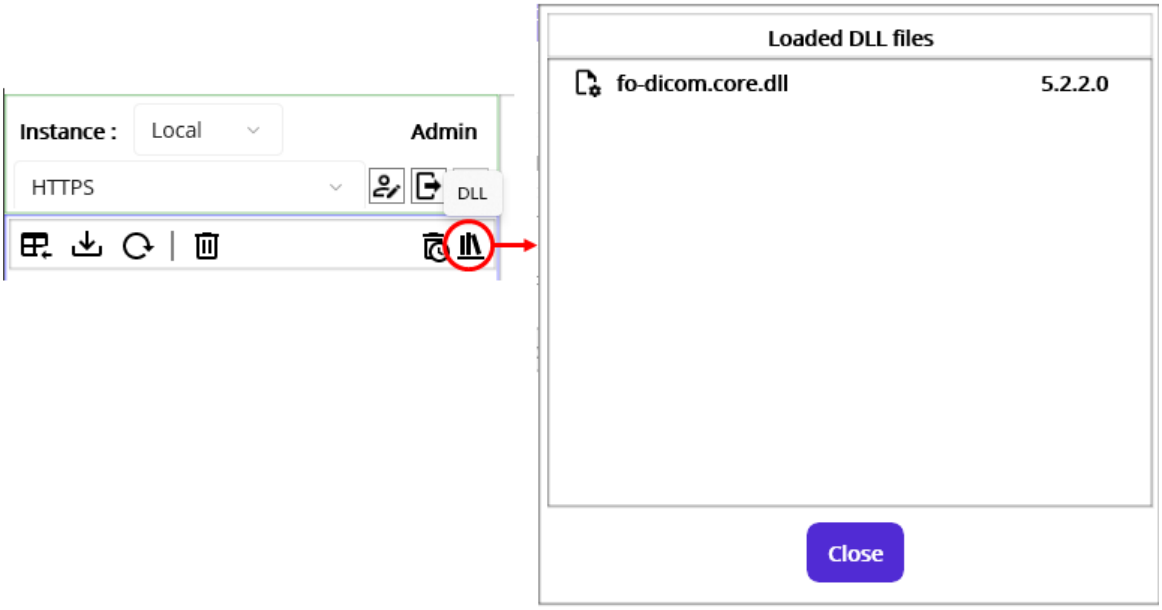


Figure 89.Loaded DLL popup

For now, we have fo-dicom.core.dll file only. So, we need to put required "CommunityToolkit.HighPerformance.dll" in the DLL folder. **Switch to the Development workstation** and select DICOM Reader service. If necessary, revisit Figure 26 to see how to modify DLL. As shown in Figure 90, you can download the DLL or set Fallback value. You can choose any of the options, but we prefer setting fallback values for this tutorial. Search "**CommunityToolkit.HighPerformance**" in the Fallback tab. Select the first result and choose the proper version or the latest version, in this case "8.4.0 and NETCoreApp.Version=v8.0." Then click the Add (⊞) button.

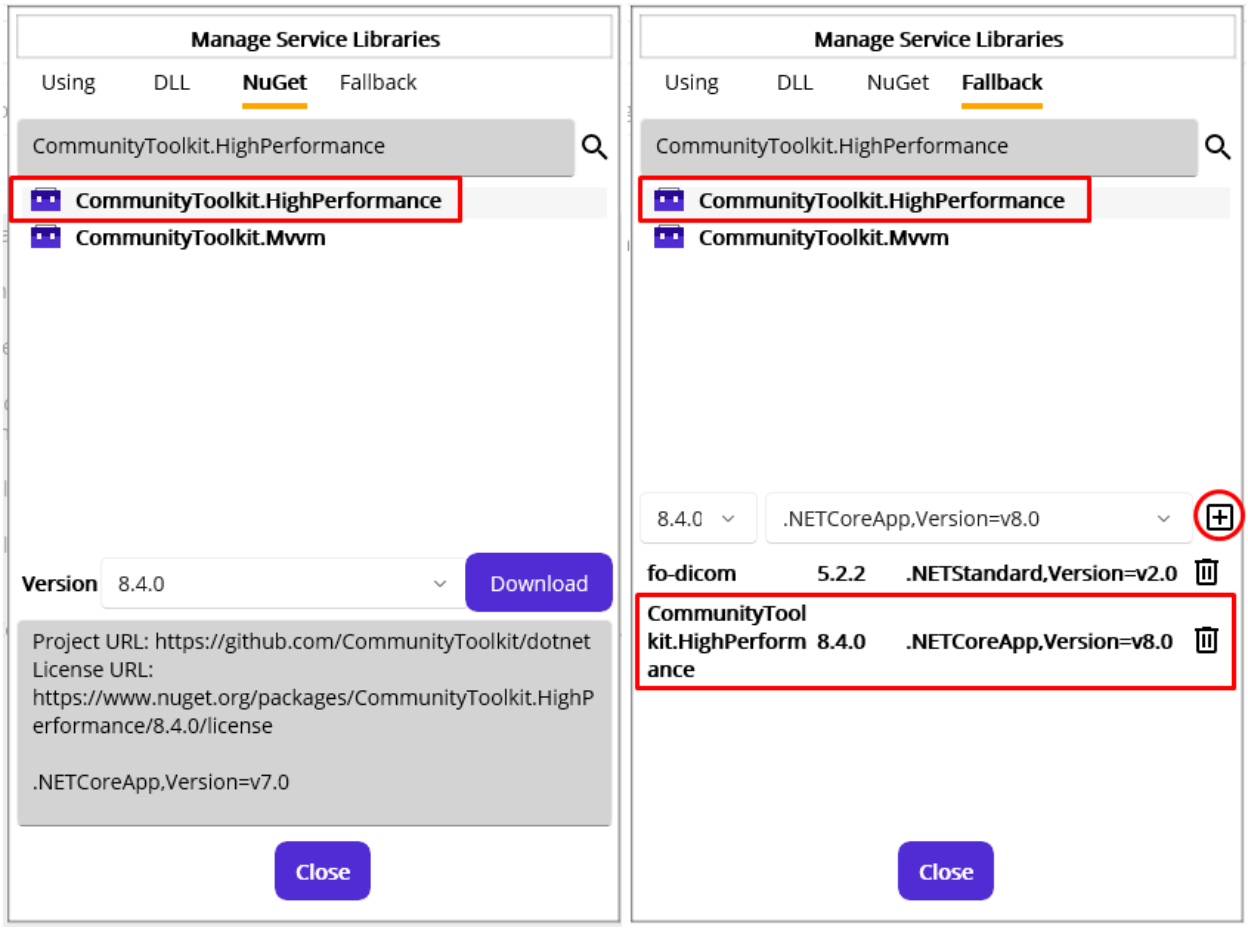


Figure 90.Download or set Fallback DLL

Once you set fallback value, click the Save ( ) button to save the DICOM Reader service and click the Load ( ) button.

If the load process is finished, **switch back to the Management workstation** and click Source tab.

Click the Refresh Services ( ) button to refresh the server to see the up-to-date server files. Like we saw the fallback values in Figure 71, select the DICOM Reader and click the NuGet Fallbacks button to check the updated fallback values on the detail view, as shown in Figure 91.

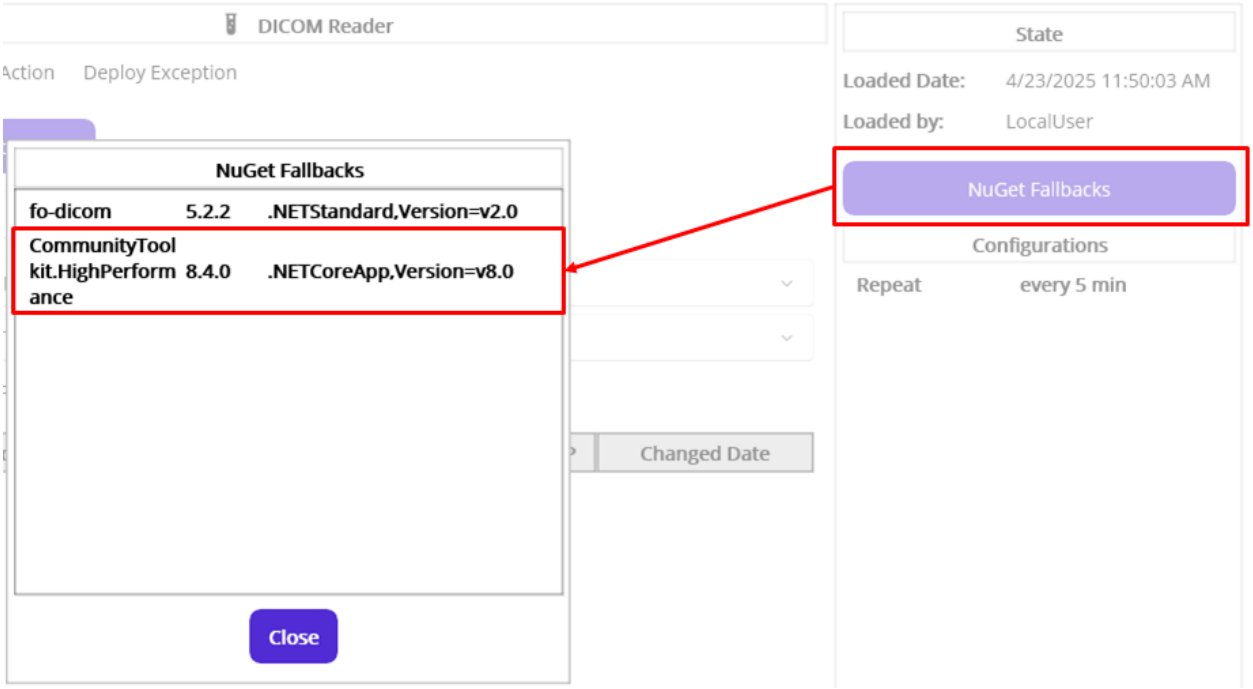


Figure 91.NuGet Fallbacks popup showing added DLL info

We saw that the newly added DLL info, CommunityToolkit.Highperformance, is in the DICOM Reader. Now let's deploy the service again.

Note: The Macaron server is open to the other Damia as long as the user is verified with a valid credential. And the other logged-in user can modify files on that server while you are working with it. If you have opened the management workstation for a long period of time and there's any possibility that someone touches the server, please refresh the server frequently.

Since there is no other model is modified, keep DICOM Reader selected and click the Deploy ( ) button. And click the Yes button to proceed with the deployment, as shown in Figure 92.

Figure 92.Deployment confirmation popup

At this time, the deployment process may take a little longer than before since that involves DLL file download. When we initiate the deployment, the server checks the fallback values and downloads the specified DLL from the NuGet server if that doesn't exist.

Note: Whenever the server starts, the server checks each service's fallback values and downloads the specified DLL files if they do not exist, and then copies them to the DLL folder. Downloaded files will be kept and reused later instead of being downloaded again.

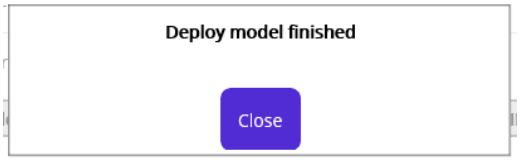


Figure 93.Finished message popup

When you see the deployment is finished popup, as shown in Figure 93, close that. As an extra verification step with the modified service, check the server side DLL files by clicking the DLL ( ) button, as shown in Figure 94.

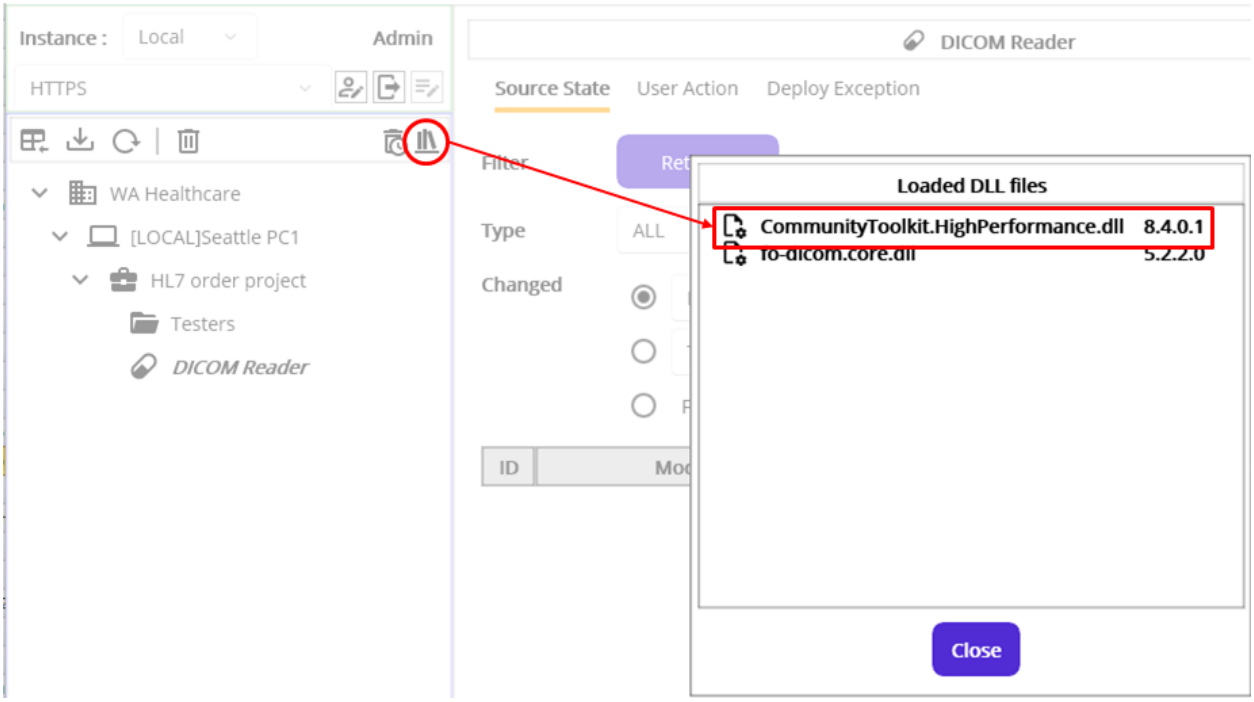


Figure 94.Downloaded DLL after the deployment using fallback values

As we had a missing DLL error in the last transaction, click the Observe > Transaction tab to look at the new transaction record. Without selecting any options, click the Retrieve button. At this time, you will see Succeed in the Type column, as shown in Figure 95. Click the Exception ( ) button to see if there's any error message.

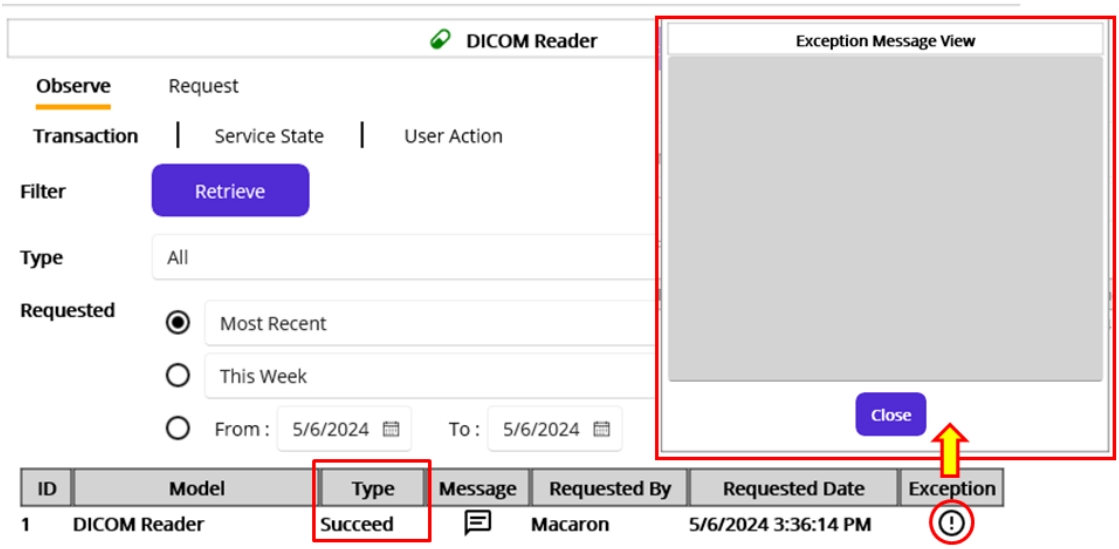


Figure 95.The latest record with Succeed type.

As you see, there is no ERROR message in the popup, as shown in Figure 95. Now we can check the location where we put the sample DICOM files and result txt files after each execution of the DICOM Reader service. Briefly mentioned earlier, a sample DICOM file obtained from NEMA site was copied to Dicoms folder for this test. Open File Explorer and locate the Dicoms folder, as seen in Figure 96.

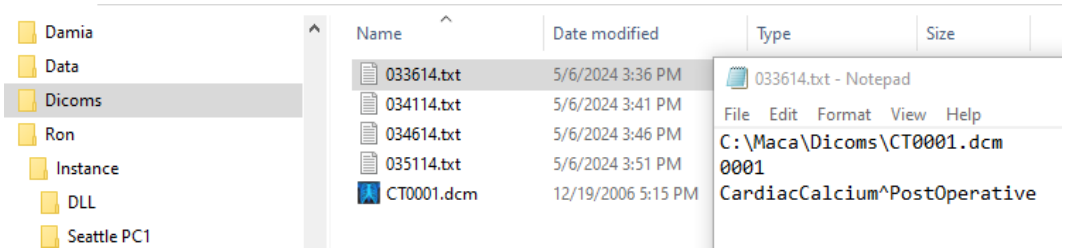


Figure 96.Generated txt files in the Dicoms folder

Since we set the DICOM Reader to execute the code every 5 minutes, some txt files had been generated for the last 20 minutes. Among them, open the first one, which is 033614.txt, to see the data we tried to extract using the code defined in Figure 45. As you see, the text file contains three rows of data; 1) File information, 2) Patient ID, and 3) Patient name. The key part that we will use is patient's ID and name. So, we can reuse the related code to compose HL7 message in the Development workstation to extend the code in the DICOM Reader service. We will go over that part later.

Service State Retrieval

Along with the transaction history, we can track the service's state history in the Service State view. One thing to note is that this view shows Macaron Service data only. As mentioned earlier, **a group model can't be started or stopped**. For this reason, there is no concept of a Started or Stopped in the group model, and they won't be retrieved here. As we did in the above sections, we can select a group model to retrieve all containing services' states. Or, we can just select the service model to focus on the specific service's history. Similar to the Source State Retrieval view, this shows the result of the user actions, such as Start or Stop commands. In addition to that, this shows the results after the Macaron server started. You can track that information based on the selected time frame using the filtering options shown in Table 9.

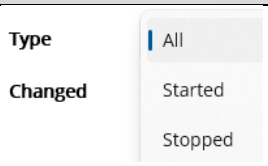
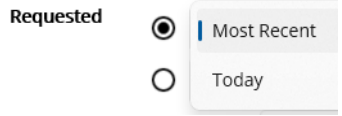
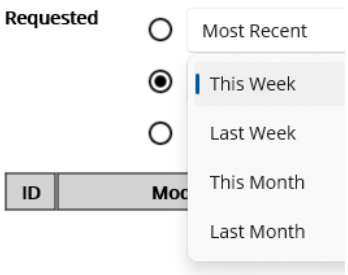
Filtering options	Details
	**Deployed Service State Types** • **Started**: Deployed Macaron service was started. • **Stopped**: Deployed Macaron service was stopped.
	Along with the selected type above, you can retrieve the most recent records or today's records.
	Along with the selected type above, you can retrieve a week's or a month's records.
	You can define specific time frame with this option.

Table 9.Service State filtering options

At this time, click the This Week radio button to retrieve all changes that happened this week. Based on the timeline, the first deployment was done yesterday, and the modified service deployment was done today. Since we didn't stop the running DICOM

Reader service, deploying selected DICOM Reader stops the running service first, then deploys (starts) the latest service file. Generated two records explain the result of the deployment in Figure 97.

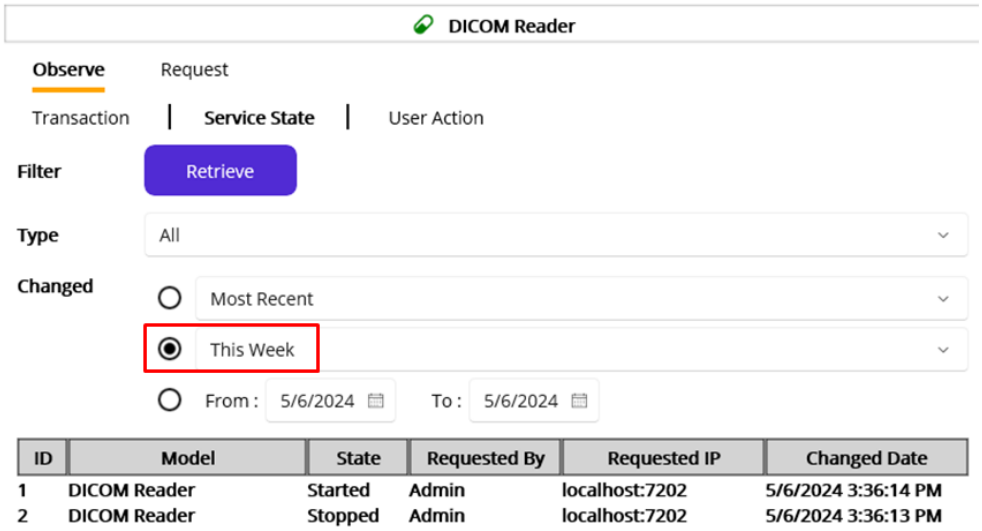


Figure 97.Retrieved DICOM Reader's state change history

As you may notice, yesterday's record with Started state is not in the result even if we selected This Week. We still have the record somewhere in the database, and the result supposed to look like Table 10.

ID	MODEL	State	…	Changed Date
1	DICOM Reader	Started	…	5/6/2024 3:36:14 PM
2	DICOM Reader	Stopped	…	5/6/2024 3:36:13 PM
3	DICOM Reader	Started	…	5/5/2024 5:43:53 PM

Table 10.Service state history of the DICOM Reader in a database

Session based record displaying

It is not related to a web-based session and more like a conceptual meaning to define whether the service was deleted or overwritten with a modified one, and disable old one accordingly. As we loaded modified DICOM Reader service on the server from the Development workstation, previously loaded DICOM Reader's information kept in the database marked as Deactivated and all associated historical records, such as transaction data, are unavailable. Since the newly loaded DICOM Reader is assigned a new internal ID, like a token of an active session, all new transaction data from the last loaded date will be associated with that ID. This helps you to focus on the lastly loaded model's data by filtering all previous models' history out. For this reason, 3rd record in Table 10 was recognized as old session's data and automatically filtered out.

This feature is by design, but still, there are some disadvantages, like can't retrieve what happened before. For example, if you try to revisit the error produced by old code for debugging or just reference purpose, there's no way to view that data once you loaded modified service. Since this is a prototype version, there would be a design change in the future release if necessary.

User Action Retrieval

Click User Action tab to show User Action retrieval view, as shown in the Figure 99.

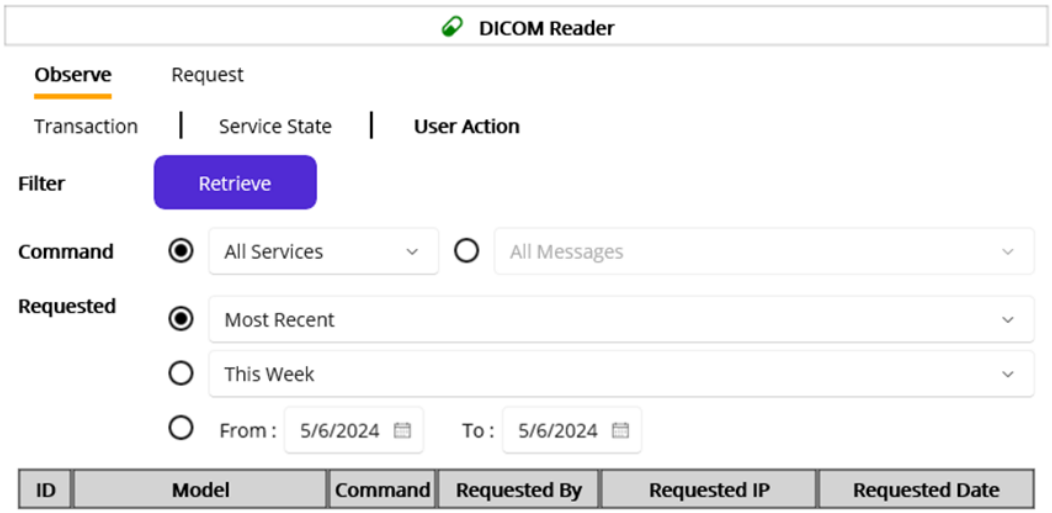


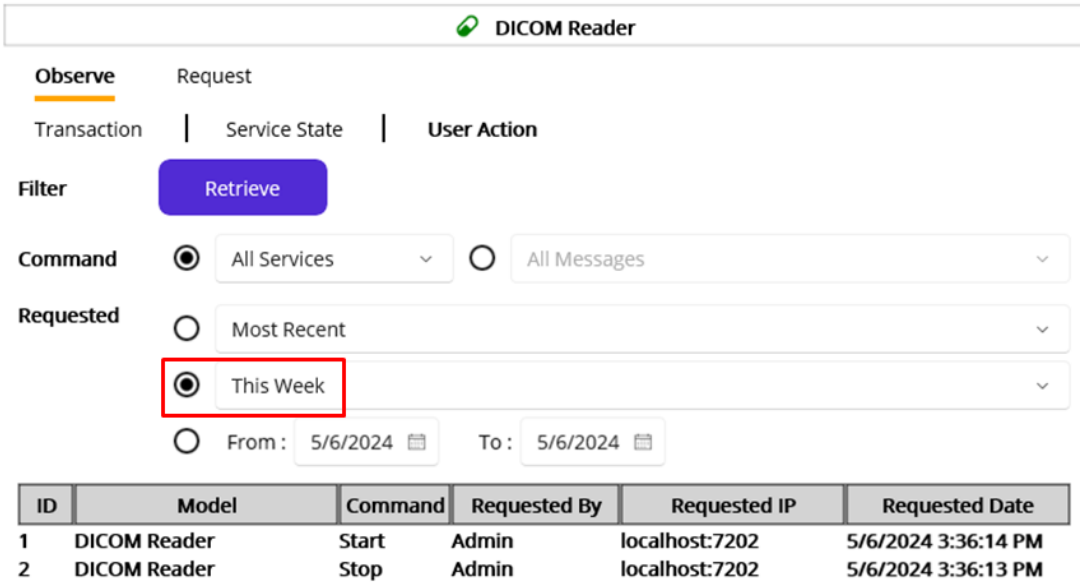
Figure 98.User Action Retrieval view

As you already familiar with the term User Action that mentioned in the other view, see **Figure 76**, this tracks user action history related to deployed services based on the following filtering options, shown in Table 10. Since the Macaron server is considered as a user as well, Macaron user's Start command, which will be initiated when the server starts, will be retrieved here. Revisit **Default Users** section and check the **Table 3** for the details.

Filtering options	Details
Command Requested All Services Start Service Stop Service Restart Service	This is associated with the Instance Model Toolbar, as shown in **Table 7**. **Service Command Types** • **Start**: When a user clicks the Start ( ) button or the server starts. • **Stop**: When a user clicks the Stop ( ) button. • **Restart**: When a user clicks the Restart ( ) button. This will involve Stop and Start processes.
Requested Most Recent Today	Along with the selected command type above, you can retrieve the most recent records or today's records.
Requested Most Recent This Week Last Week This Month Last Month ID	Along with the selected command type above, you can retrieve a week's or a month's records.
From : 5/5/2024 To : 5/5/2024	You can define specific time frame with this option.
All Messages Process Message Reprocess Message Retrieve Message View Message To :	**Message and Request Command Types**(*This is not implemented*) This option is targeted to retrieve Message and request related user actions. If a user retrieves the record or view message data, Maca saves that action and the history will be retrieved with this option.

Table 11.User Action filtering options

Once you see the User Action view, select the This Week radio button, and click Retrieve button. The following Figure 99 shows almost the same data compared to Figure 97. One thing you need to check is the 2 records with Stop and Start commands. Although the Requested By column is populated with "Admin," we never clicked any of Start, Stop, or Restart button in the Instance Model Toolbar. See the Table 7 for the details. Those records were generated by the Deploy command from the Source tab after we modified the DICOM Reader with Fallback values. When the target service, in our case DICOM Reader, is running ( ), the Deploy command will undeploy(stop) the service first, and then initiate the Start command. As a result, these two records were generated, as shown in the Figure 99.

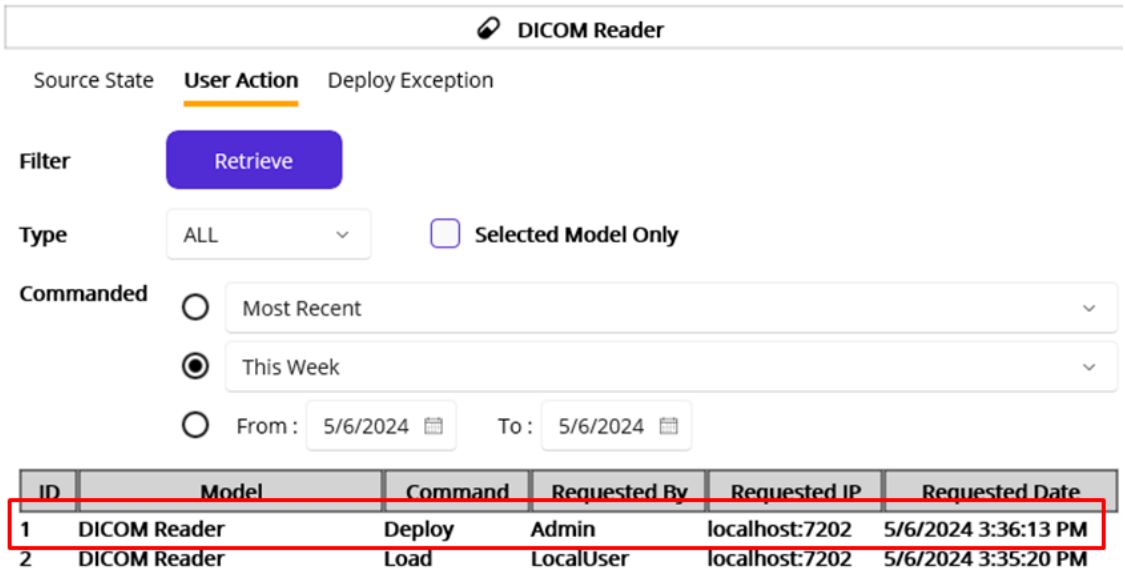


ID	Model	Command	Requested By	Requested IP	Requested Date
1	DICOM Reader	Start	Admin	localhost:7202	5/6/2024 3:36:14 PM
2	DICOM Reader	Stop	Admin	localhost:7202	5/6/2024 3:36:13 PM

Figure 99.User action history of the DICOM Reader service this week

Cross-referencing user actions between deployed model and source model

Since we never clicked Start or Stop button in the Instance Model Toolbar, those records in Figure 99 need to be cross-referenced by checking User Action view in the source area. **Click Source tab** to show source model views. As the Deploy ( ) command was initiated in Source Model Toolbar, the historical record of the Deploy needs to be retrieved in the Source area. Select the DICOM Reader node and click User Action tab. Select the This Week radio button and then click the Retrieve button.



ID	Model	Command	Requested By	Requested IP	Requested Date
1	DICOM Reader	Deploy	Admin	localhost:7202	5/6/2024 3:36:13 PM
2	DICOM Reader	Load	LocalUser	localhost:7202	5/6/2024 3:35:20 PM

Figure 100.User action history of the DICOM Reader source this week

Since we didn't specify the action (command) type, all commands records about the DICOM Reader were retrieved, as shown in Figure 100. And the 1st record shows the Deploy command history that actually triggered Start and Stop records in Figure 99. For your reference, throughout this tutorial, the data in a result table in each Figure will be displayed in descendant order, i.e., the most recent record will be shown in the top row. As a side note, right below the Deploy, there is a Load command related record that eliminates old records.

Note: As we went over the session based record display, the same logic was applied in this view. We deployed entire models, including domain and service, from the Development workstation earlier. And then the modified DICOM Reader file was loaded and deployed. Since this serial command history is saved in a database, the result in Figure 100 could be something like Table 12. However, the 2nd Load command assigns a new ID to the DICOM Reader, all command history before the last load was marked as Deactivated and filtered out. Then the new records created after that will be recognized as an active data, as shown in Figure 100. Again, this is by design for now, and the functionality would be changed in the next release if necessary.

ID	Model	Command	Requested By	Requested IP	Requested Date
1	DICOM Reader	Deploy	Admin	Localhost:7202	5/6/2024 3:36:13 PM
2	DICOM Reader	Load	LocalUser	Localhost:7202	5/6/2024 3:35:20 PM
3	DICOM Reader	Deploy	Admin	Localhost:7202	5/5/2024 5:43:53 PM
4	DICOM Reader	Load	LocalUser	Localhost:7202	5/5/2024 3:22:07 PM

Table 12.Command history of the DICOM Reader in a database

We checked the Deploy command history, now let's cross-check the Source State history because the command changes the related service's state as well. Click the Source State tab and select the This Week radio button. Then click the Retrieve button to see if the result contains Undeployed and Deployed records. Since the Deploy command undeploys the service if it is deployed, there is a record with Undeployed State. Then the Deployed state is at the top row, as shown in Figure 101. Likewise, if there is no deployed service, only a Deployed record will be found. Last but not the least, the Source State will filer the old record out when new model is loaded. So, the 3 records in Figure 101 are the records after the last DICOM Reader load from the Development workstation.

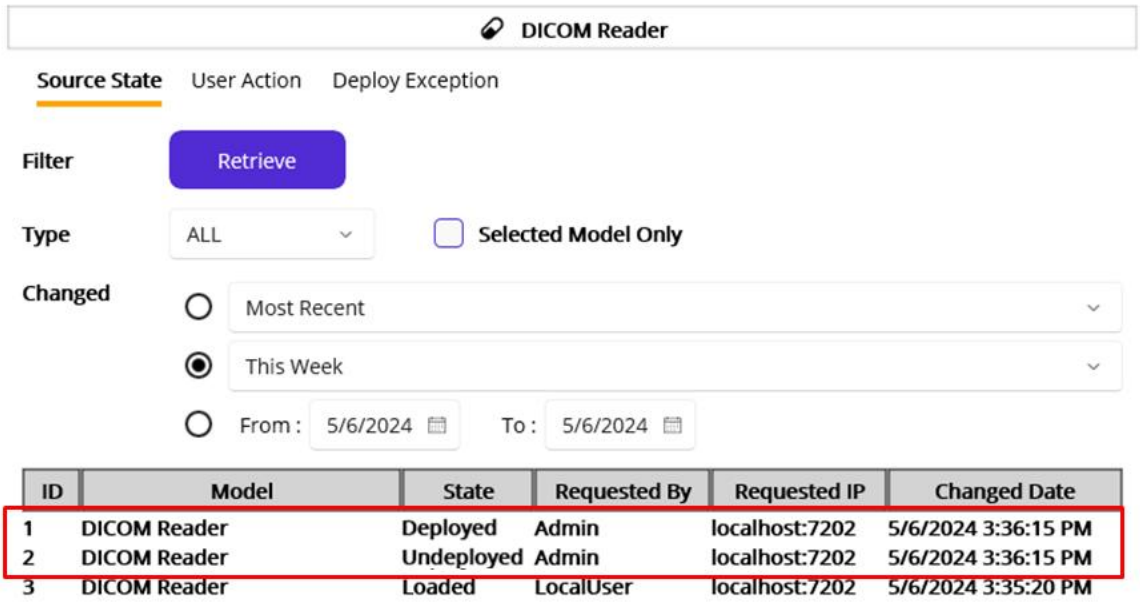


Figure 101.Source state history of the DICOM Reader

As briefly mentioned earlier, the Source State view shows the result of the command. If any of the commands are not complete, no command-related result will be found in the Source State view.

Request Instance Model

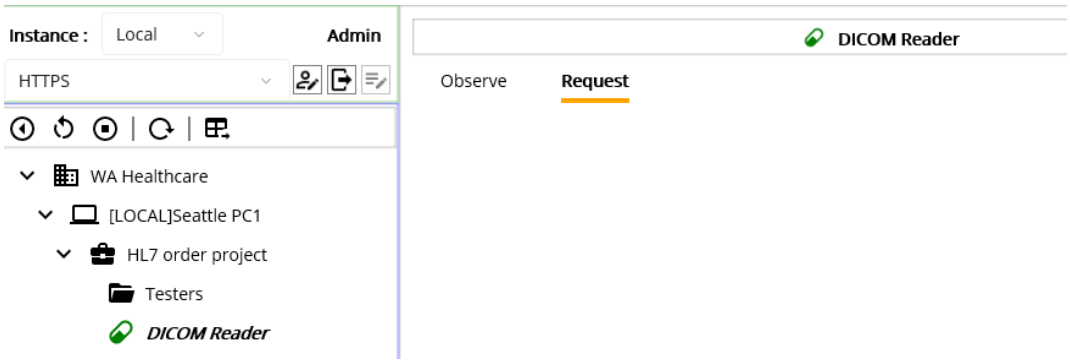


Figure 102.Request tab

Once you click the Request tab, you will see nothing as you see in Figure 102. Before we delve into the Request tab, let's recall the process of creating a service. If you look at Figure 15 or click the create project model button, you will notice that there are 4 service types. Although we haven't touched RequestListener, HL7Client, and HL7Listener services yet, it is a good time to think about how each service is triggered.

First of all, we just implemented the DICOM Reader service, which is a CodeRepater service type. As the name describes, the CodeRepeater executes the code in a timely manner, i.e., based on the time interval or on the defined specific time. So, there is no need to trigger the service to execute. But in many cases, we need services that interact with incoming external requests, i.e., execute the code when it was triggered. This is where the Request comes from.

Note. The term "external" here doesn't mean outside of the Maca. From one MacaronService's standpoint, the other MacaronServices inside of Maca are external. For example, from the ORM Sender's perspective, the DICOM Reader is external. And of course, outside of the Maca is also external.

As a reminder, one of the objectives of the project is "*sends HL7 order messages to the client's application*." We extracted the required information from the DICOM file, and the next step is sending the HL7 message. We may implement a sending feature inside the DICOM Reader, but it would be a complicated process because of the TCP/IP communication with the external application, not to mention the code management. In that case, we can create a standalone HL7Client service that manages TCP/IP communication. And let the DICOM Reader trigger the service to execute its code by sending the composed HL7 messages. As you may notice, there is no human intervention in terms of requesting. Once we set the request flow, i.e., **Request Mapping**, the Macaron will process the requests. We will cover this in detail with actual code later, but here's the brief flow of the request with the HL7Client.

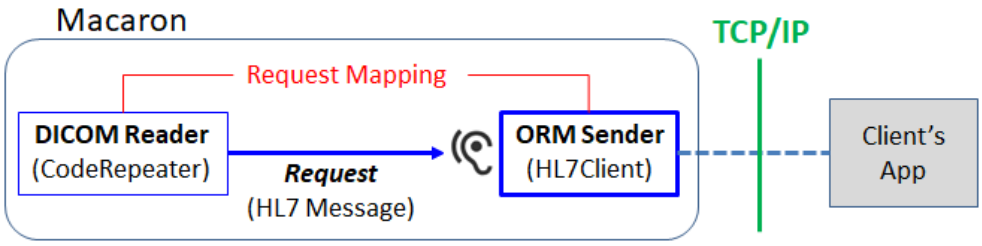


Figure 103.Basic request flow

Although it is a good approach in terms of separation of concern, keeping two services just for code maintenance is still not enough for the service development perspective. Like other application design methodologies, this approach can provide extra features, such as sending custom HL7 messages on the fly, resending previous messages, and processing multiple messages sequentially for testing.

Note. By design, HL7 message resending is not implemented in the prototype version because of the possible sensitive data breaches related to HIPAA compliance. We will cover this topic later in the HL7 data security section. So, the "Reprocess Request" in the Figure is for instructional purpose only.

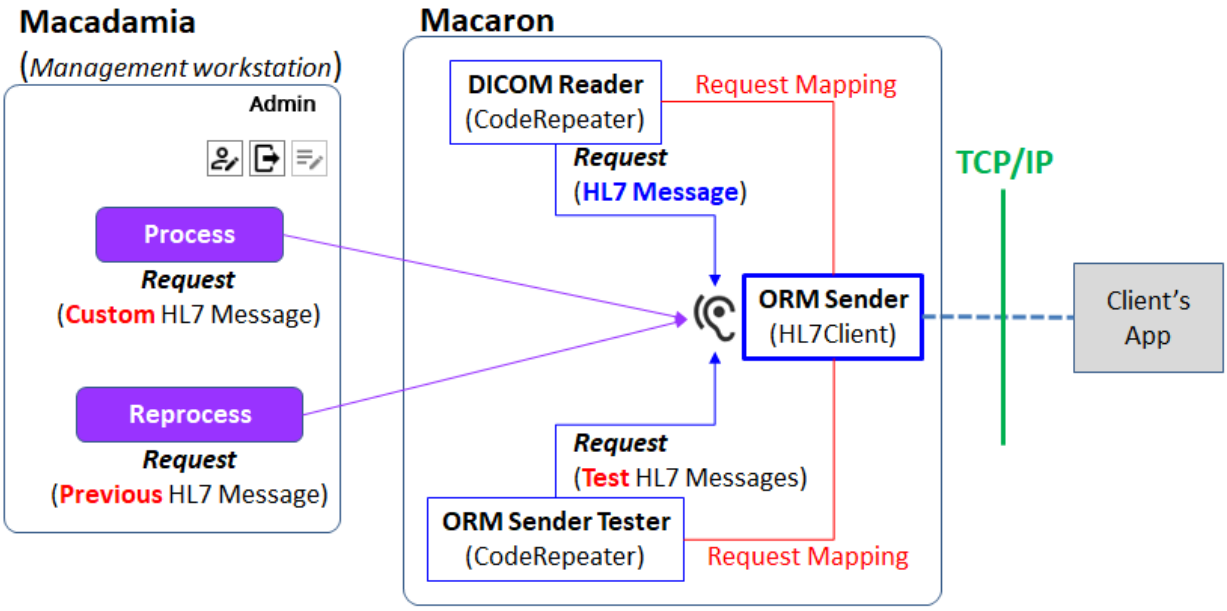


Figure 104.Complete request flow

As you see in Figure 104, the ORM Sender is not dedicated to a specific service. As long as the Request Mapping is done or any user-driven requests come, it will process the incoming messages with the code defined in the service. So, we will call the service listening to external requests "**Requestable Service**." As a side note, the ORM Sender doesn't return any value(result) to the caller because it uses the queue object to send the message sequentially to the target application by design. The ORM Sender checks the queue and deques the message if exists and then process the message and send to the client app if there's no issue with the message.

Secondly, the RequestListener service that we will cover later is another requestable service. But this service expects custom arguments instead of HL7 messages. So, we can request the RequestListener though Macadamia's Management workstation as well. We will cover this one later in detail. Below Table 13 is the comparison of each service's execution type.

Service Types	Requestable	Expects	Executes the code when:
CodeRepeater	No	None	The time interval or defined time(s) is reached
RequestListener	Yes	Arguments	The arguments have come
HL7Client	Yes	HL7 message	The HL7 message has come
HL7Listener	No	HL7Client or external App	A message through TCP/IP has come

Table 13.Execution types based on the Service Types

Once you have a requestable service, you can map request using the Add Service Request Code ( ) button in the calling service, in this case the DICOM Reader. The popup will show the RequestListener and HL7Client services only, and you can select a requestable service to map the request. This will be covered later soon.

Now back to the original topic, the Request tab here shows the request UI only when the RequestListener or HL7Client service is deployed and selected on the project explorer. So, you need to create and deploy one of them first. Then you can type any custom HL7 message and click the Proces button to send the request to the HL7Client. In addition, you can retrieve the transaction of the HL7Client service and check the message button to see the received message. But due to the security issue stated above, Maca saves the MSH segment value only instead of the entire message in the database. So the popup doesn't have a Reprocess button. This will be demonstrated later as well.

So far, we have gone over the entire process, from the service development to the deployment, along with quick debugging. Since we wrote a few lines of code to test, there are still many parts that we didn't touch. By extending the code of DICOM Reader service and adding more Macaron services, we will cover various situations. Let's get back to the scenario and continue the objectives.

Build HL7 messages

As we wrote a code to parse the DICOM file and extract values from the Tags in the DICOM Reader service, the next step is composing an HL7 message, in our case order message, with extracted values. Before we compose the message, let's go over the HL7 message protocol real quick.

Health Level Seven (HL7)

Introduction

HL7, stands for Health Level Seven, is a standard message structure in exchanging medical and administrative data between healthcare systems. As the term exchanging implies, once a message is sent, an acknowledgement message is transmitted back as a response. To facilitate this, Maca provides two built-in services, HL7Client and HL7Listener. These connection-based services handle sending and receiving HL7 messages by communicating between them or with external applications. In our case, you will send an ORM message, which is an order, and the client will send an ACK message, which is a response. We will go over the ORM and ACK shortly in detail. Following Figure 105 shows possible implementation of the message exchanges between Macarons or with external Interface (Integration) Engines, such as Mirth Connect[21] or Infor.[22]

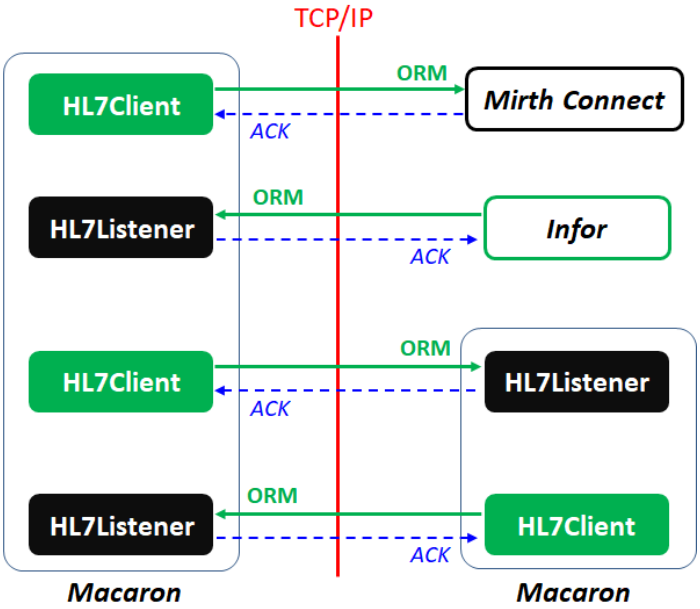


Figure 105.HL7 Message exchange implementation examples

Helpful websites to start with

The first web site you may start with is HL7 International[23]. You can sign up to get more in-depth material about HL7 version 2 messages that we will implement. In addition, you can check each message's detail value sets by checking its HL7 Code Systems (https://terminology.hl7.org/codesystems.html). As Maca supports v2x, you can check out V2 Code Systems (https://terminology.hl7.org/codesystems-v2.html), which is a huge list but will be very helpful when you are building and parsing HL7 v2.x messages. Along with that, some handy sites will be introduced as an alternative. You will see some examples excerpted from the sites in the next sections.

Hl7 Message versions

For decades, v2.x HL7 messages were used and are still popular in many systems. Meanwhile, v3 was introduced, but Macaron's built-in HL7 services support v2.x only. As v2.x grows, there are minor changes but in most cases they share most of the structure.

[21] https://www.nextgen.com/solutions/interoperability/mirth-integration-engine
[22] https://www.infor.com/products/cloverleaf
[23] http://www.hl7.org/

However, it is recommended to check each version's structure when you work with other systems that use different versions. One helpful site is a HAPI[24], which is a Java based HL7 API that support v2.1 to v2.8.1, as shown in Figure 106.

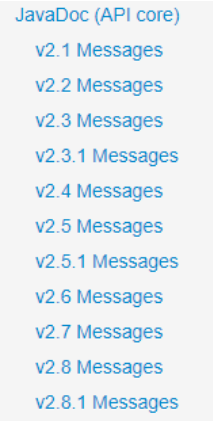


Figure 106.HL7 v2.x message list in HAPI

You can check each version by clicking any of the links on the HAPI website. As mentioned, HL7 org will provide thorough information, but HAPI will provide handy information that you can refer to.

HL7 Message types

It would be quite a long time to list all HL7 message types for each version here. Instead of that, let's focus on the order message in v2.3.1. If you click the "v2.3.1 Message" link (https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/index.html) from the HAPI site, you can find all types of v2.3.1 messages there, as shown in Figure 107.

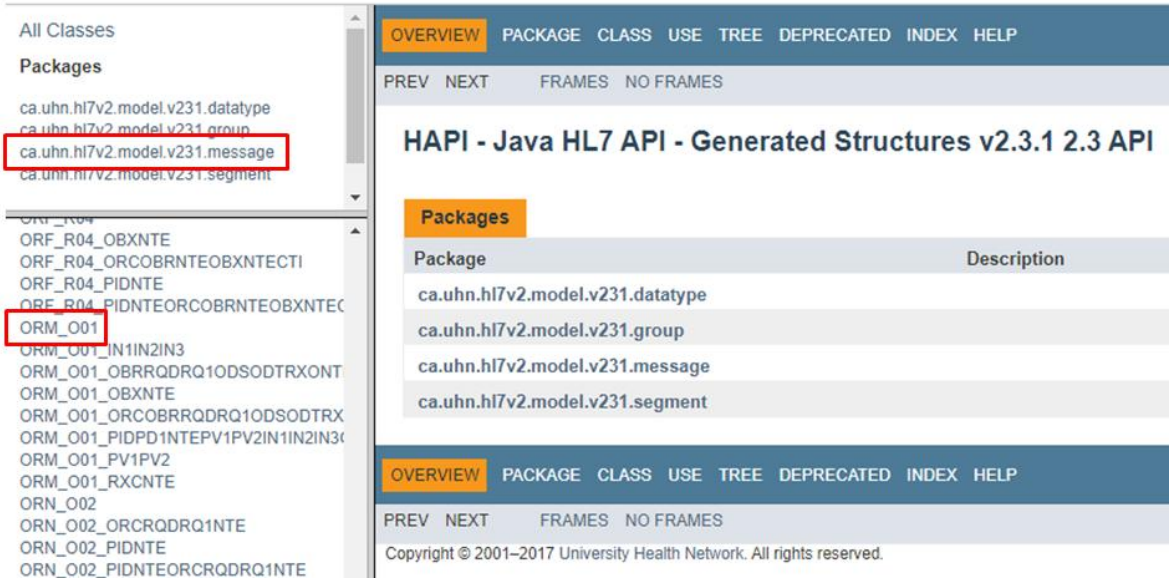


Figure 107.HL7 v2.3.1 details in HAPI

In our case, we need the ORM and ACK messages. Please note that ORM is a simplified name of ORM^O01 or ORM^O01^ORM_O01, which is used to contain order related information. As you see in Figure 107, it can be interchangeable with ORM_O01. We will go over little more on this value in the Field and Component section below. Browse the HAPI site or visit the following link to see the ORM_O01 message structure.

https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/message/ORM_O01.html

ACK can be found here; https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/message/ACK.html

[24] https://hapifhir.github.io/hapi-hl7v2/

Sample HL7 Order Message

The HL7 website provides some sample HL7 messages, including the order message. You can visit following link to check those messages; https://confluence.hl7.org/display/OO/v2+Sample+Messages

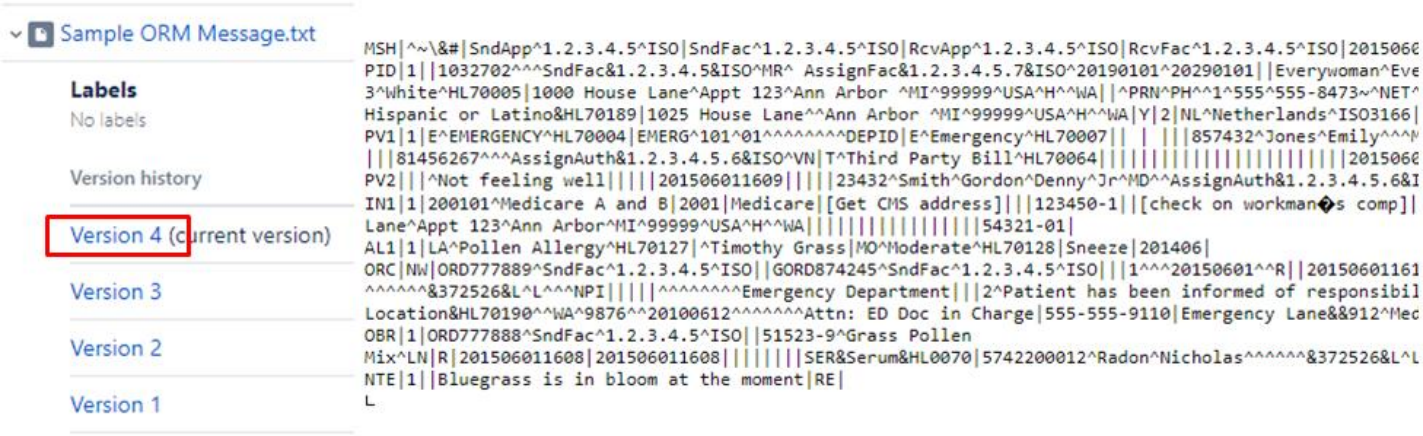


Figure 108.Captured sample ORM^O01 message

Like Figure 108, it could be overwhelming if you look at the ORM, ACK, or other message for the first time. Since we will use the helper API to parse (read) and compose HL7 message, just look each message briefly and focus on the elements that compose the HL7 message. In addition, as you may notice, HL7 message is a plain text with some special characters, such as "|^~\&" if you look into it. We will go over the elements and the special characters in the next section.

HL7 Message structure

A message is composed of the following 4 key elements: Segment, Field, Component, and Subcomponent. We will use above ORM message to demonstrate the HL7 message structure parsing. Since the message shown in Figure 108 is part of the entire message, please visit the link to see the entire message first. As briefly mentioned, the HL7 message can be parsed with a helper API, which is HL7-V2[25], all those element could be accessible with it. So, please get used to identifying Segments, Fields, Components, and Subcomponents, which will give you a faster development speed with the helper API.

Segment

Segment is a group of field starts with segment name, such as MSH, PID, and PV1. Following value is an example of MSH (Message Header) segment. As a side note, the given sample message has an extra value (#) in the delimiters area like "**|^~\&#**" and removed because the normal value should be "**|^~\&**" in the MSH segment like below. And the delimiters will be covered in the following sections.

```
MSH|^~\&|SndApp^1.2.3.4.5^ISO|SndFac^1.2.3.4.5^ISO|RcvApp^1.2.3.4.5^ISO|RcvFac^1.2.3.4.5^
ISO|20150601160901.12+0100|20150601160810+0500|ORM^O01^ORM_O01|5381904|P|2.3.1|||AL|AL|US
A|ASCII|en-US^^ISO639|
```

Note: Every HL7 message must start with the MSH segment. If a message, i.e., received data, does not start with MSH, it will be considered an invalid HL7 message.

And following value is an example of PID (Patient Identification) segment.

```
PID|1||1032702^^^SndFac&1.2.3.4.5&ISO^MR^
AssignFac&1.2.3.4.5.7&ISO^20190101^20290101||Everywoman^Eve^L^Jr^Dr^^L^^^^G^20000909^2030
1231^PhD~ Original^Eve^L^Jr^^^M^^^19700601&20000908^G||197006010912|F||1002-5^American
Indian or Alaska Native^HL70005~2106-3^White^HL70005|1000 House Lane^Appt 123^Ann Arbor
^MI^99999^USA^H^^WA||^PRN^PH^^1^555^555-
8473~^NET^Internet^eve@test.test|^WPN^PH^^1^555^555-1126^12|en-
US^^ISO639|M^Married^HL70002|CHR^Christian^HL70006|12345^^^
```

[25] https://github.com/Efferent-Health/HL7-V2

```
SndFact&1.2.3.4.5&ISO^AN||12345^MI^20180219||N^Not Hispanic or Latino&HL70189|1025 House
Lane^^Ann Arbor ^MI^99999^USA^H^^WA|Y|2|NL^Netherlands^ISO3166||||N|
```

In addition, **the segment can repeat**. So, you may see the multiple segments with the same segment name, like below.

```
OBX|1|NM|6153-1^IgE Blue Grass Kentucky^LN|1|3.9|kU/L|<0.10|A^Abnormal^HL70078||N^None -
generic normal range^HL70080|F|||201506011608|CentralLab^Central
Laboratory^HL70624|1234^Observer^Test^^^^^^LabFac&8.7.6.5.4&&ISO|||201506011605|||||||||
RSLT
OBX|2|NM|6041-8^IgE Bermuda Grass^LN|2|0.59|kU/L|<0.10|A^Abnormal^HL70078||N^None -
generic normal range^HL70080|F|||201506011608|CentralLab^Central
Laboratory^HL70624|1234^Observer^Test^^^^^^LabFac&8.7.6.5.4&&ISO||||201506011605|||||||||
RSLT
OBX|3|SN|6265-3^IgE Timothy Grass^LN|3|<0.10|kU/L|<0.10|N^Normal^HL70078||N^None -
generic normal range^HL70080|F|||201506011608|CentralLab^Central
Laboratory^HL70624|1234^Observer^Test^^^^^^LabFac&8.7.6.5.4&&ISO||||201506011605|||||||||
RSLT
```

Above is a part of sample ORU message in https://confluence.hl7.org/display/OO/v2+Sample+Messages. As you may see later, HL7-V2 API always returns found segment as a list in this reason.

Note: HL7 message is a plain text. What you will see in the received message is wrapped text like above OBX segments. Since an HL7 message can have a long line of text, each segment couldn't be easily identified. To make the segment recognizable, **each segment has a carriage return at the end of the segment**. It is quite tricky to find the segment with your eyes. With a help of the code-base parsing, each segment could be extracted from the text. Please note that a carriage return (CR) is ASCII 13, which can be inserted as **0x0D**, **'\u000d',** or **"\r"** in the code editor.

These segments are quite complicated to look at, but when you read the message by yourself, take a closer look at the segment name first. That will give you a chunk of message to divide and parse. That chunk is a group of fields that will be covered in the next section.

Field

Field is a value separated by "**|**" delimiter in a segment. Following PID segment contains fields in red.

```
PID|1||1032702^^^SndFac&1.2.3.4.5&ISO^MR^
AssignFac&1.2.3.4.5.7&ISO^20190101^20290101||Everywoman^Eve^L^Jr^Dr^^L^^^^G^20000909^2030
1231^PhD~ Original^Eve^L^Jr^^^M^^^19700601&20000908^G||197006010912|F||1002-5^American
Indian or Alaska Native^HL70005~2106-
```

Like Figure 109, the HAPI site provides a list of the segments that you can look at. Once you click the target segment, e.g., PID, it shows the field list for the segment, as shown in Figure 109. As you see, each field is assigned a number, and that will be used as an index for accessing fields with HL7-V2 API. You can browse the site to look at more or visit the direct link for the PID: https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/segment/PID.html

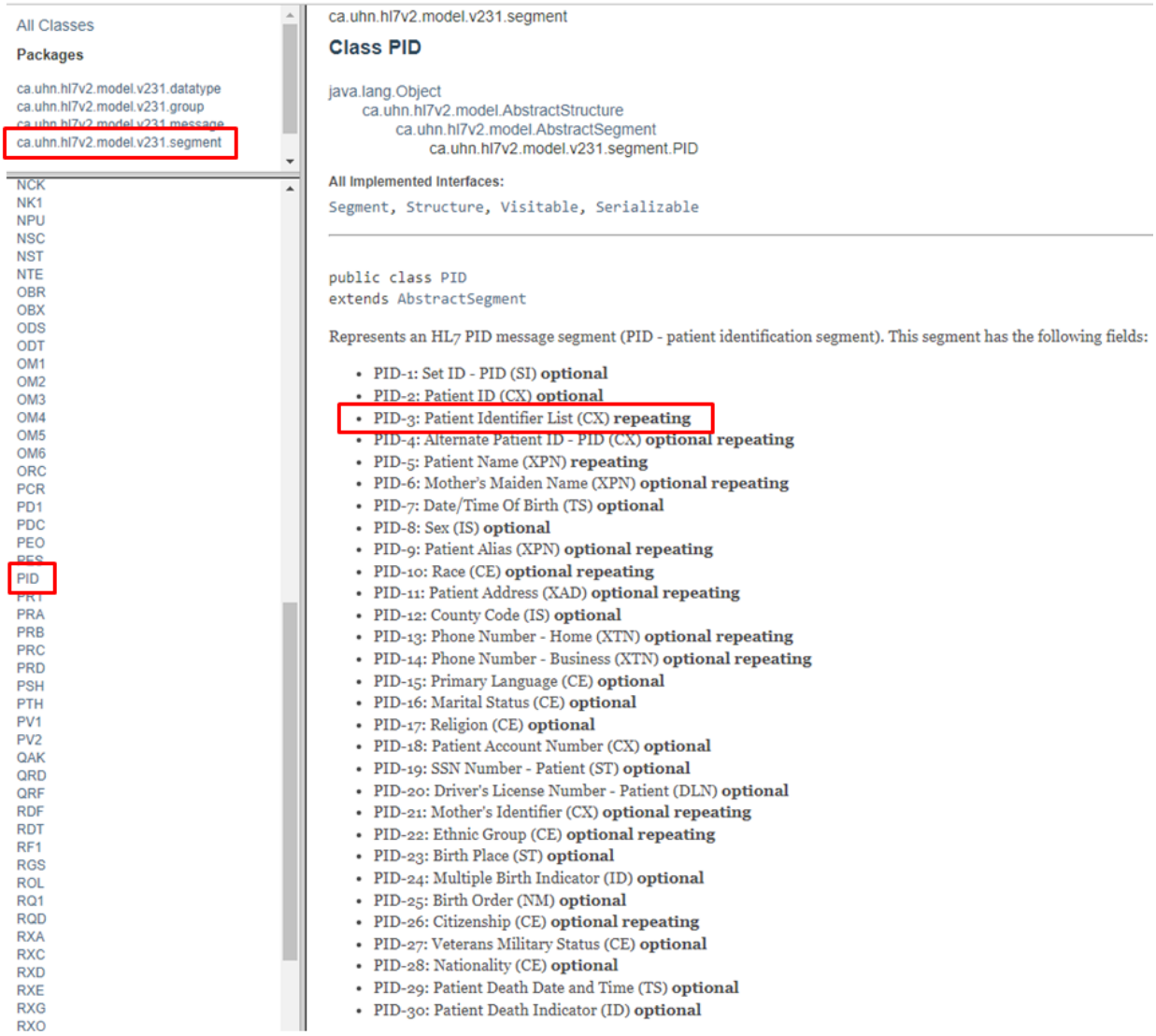


Figure 109.PID segment and its fields

As we extracted patient information from the DICOM file in the DICOM Reader service, we can use that information to write a new ORM message. Revisit Figure 45 for the code. Based on the fields in the PID segment in Figure 109, we know that PID-3 field is a place for the list of Patient ID, and the PID-5 field is a place for patient name. From now on, we will call it simply PID-3 instead of PID-3 field. So, what we have to do to compose ORM message inside the DICOM Reader is to map these values.

Note: Each field definition in Figure 109 is an important part to identify components and subcomponents. When you look at the field definition, check carefully the related parts, such as ST, optional, or repeating.

Since the field that's populated with patient ID is PID-3, let's take a look at the PID-3 value in red, as shown below.

`PID|1||1032702^^^SndFac&1.2.3.4.5&ISO^MR^AssignFac&1.2.3.4.5.7&ISO^20190101^20290101||…`

As you may notice, PID-3 can have multiple patient ID data based on the field definition in Figure 109, like below colored in red.

- PID-3: Patient Identifier List (CX) **repeating**

Please note that the given sample PID-3 is not repeating. For repeating data, the value format will look like below, populated with fake data.

PID|1||123~456~780||

As you can see, the component separator (^) that we will go over later was not used intentionally to make the data more clear. And repetition separator (~) was used instead. In a field, the tilde implies that the field has multiple field values. In the above PID-3 case, 2 repetition separators means there are 3 patient id values. So, when you see each field, pay some attention to the repeating field to read them correctly.

Although the sample PID segment doesn't have PID-2 value, you will see PID-2 is populated in many cases like below sample.

PID||123456789^^^^MRN|||DOE^JOHN||19620504|M|||||||||9999|111-22-3333

Once you see the PID-2 field definition, you will realize that the field is also a placeholder for the patient ID as shown below.

- PID-2: Patient ID (CX) **optional**

But as you see, the field is defined as optional, and thus it is totally fine if the field is not populated like the sample PID segment. However, you will see some applications populate PID-2, which is not repeating, instead of PID-3. So, you need to double-check the fields when you parse incoming HL7 messages from the different applications.

Component

Component is a value separated by "**^**" delimiter in a field. One of the examples is ORM, as mentioned above. In the sample MSH that we saw in Segment section above, there is a MSH-9 field that contains message type. You can check the field list in the following link: https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/segment/MSH.html

And the 9th field is:

- MSH-9: Message Type (MSG)

As you see the red colored data type MSG, you can check the components that MSG can contain in the following link:

https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/datatype/MSG.html

And you can find the list of components in MSG data type, as shown in Figure 110, which was captured from the link.

Represents an HL7 MSG (Message Type) data type. This type consists of the following components:

- message type (ID)
- trigger event (ID)
- message structure (ID)

Figure 110.MSG data type of MSH-9 field

From the sample MSH segment, the 9th field value is ORM^O01^ORM_O01 in red, as shown below.

MSH|^~\&|…(omitted fields)…|**ORM^O01^ORM_O01**|5381904|P|2.3.1|

As the field value can be divided by the component delimiter, followed by Figure 110, we can map the value as shown in Table 14.

Component ID	Name	Value
1	Message type	**ORM**
2	Trigger event	**O01**
3	Message structure	**ORM_O01**

Table 14.Components in PID-9

Since the message type is ORM, we can call the entire message an ORM. However, in many cases, you will see that some applications call the message "ORM^O01" or "ORM_O01." It depends on the client's situation, all are interchangeable.

Note: You can check the entire Message Types in the CodeSystem: messageType(https://terminology.hl7.org/CodeSystem-v2-0076.html). Then you can see that ORM message type is for "Pharmacy/treatment order message." And the entire Trigger Events is in the CodeSystem: eventType(https://terminology.hl7.org/CodeSystem-v2-0003.html). Likewise, you can see that O01 is for "ORM - Order message (also RDE, RDS, RGV, RAS)."

As another example, take a look at the field that we saw in the Field section. The following PID-3 contains components in red.

`PID|1||1032702^^^SndFac&1.2.3.4.5&ISO^MR^AssignFac&1.2.3.4.5.7&ISO^20190101^20290101||…`

In a PID-3 field definition, you can find the field data type, which is a structure of components colored red.

- PID-3: Patient Identifier List (**CX**) **repeating**

As you can see, CX is the field's data type. So, to identify each component in a field, check the CX data type in HAPI site.

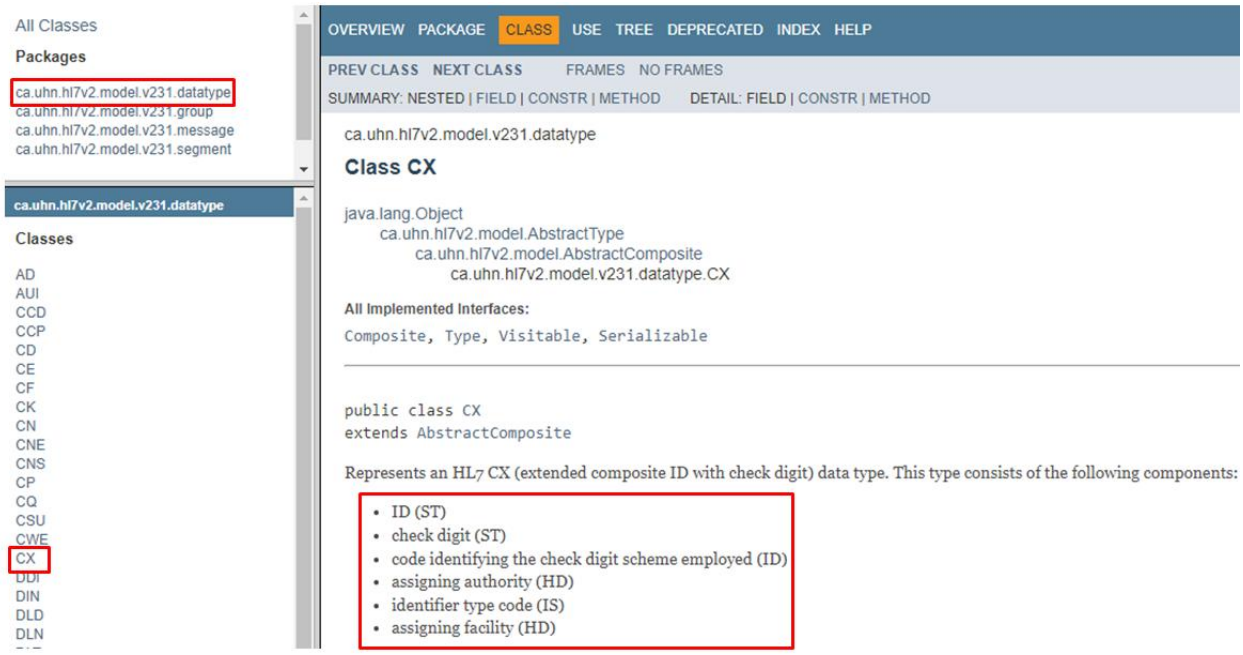


Figure 111.PID-3 components in CX data type (v2.3.1)

Based on the Figure 111, we can pinpoint the component and its value. And we can assume the index based on the appearing order. Since the component is separated by caret (^), we can divide each component, especially the above PID-3, as follow Table 15.

Component ID	Name	Value
1	ID	1032702
2	check digit	
3	Code identifying the check digit scheme employed	
4	Assigning authority	SndFac&1.2.3.4.5&ISO
5	Identifier type code	MR
6	Assigning facility	AssignFac&1.2.3.4.5.7&ISO
		20190101
		20290101

Table 15.Received PID-3 data mapped with CX data type (v2.3.1)

Note: Each field does not always have to have multiple components. As long as the receiving application does not check the rest of the components, a single value without the delimiter (^) will be accepted as well. However, a single value will always be considered the first component. For example, if the PID segment looks like "PID|1||**1032702**|…," we can get the value "1032702" with **PID.3** or **PID.3.1**. Although a single value itself is acceptable, the value may not be enough for parsing. In this case, we know the value is ID,

but we don't know whether that value is an MRN (Medical Record Number) or an application-generated ID, which could be clear if we have a PID.3.5 value. But in many cases, the receiving end already knows the value is what they expect, e.g., MRN. So, they won't check the other components, and the value will be acceptable. However, once they provide sample messages that they expect, it is recommended to populate all components populated in the sample messages.

If you look at the MSH segment above, you will notice that the HL7 version is 2.3.1. We didn't go over the MSH segment but you can visit the MSH page (https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/index.html) and check the MSH-12, version ID.

- MSH-12: Version ID (VID)

As we are on v2.3.1 based on that field's value, there is no further component than 6th one, shown in Table 15 and Figure 111. So, the last two components came out of nowhere can't be identified under v2.3.1.

HL7 version tolerance among systems

As you may work with various clients, you will face the version collision while you are interfacing the systems. Since HL7 v2x has been used for decades, not every client updates their system to use up-to-date HL7 version, which may involve their system updates that cost a lot. So, there would be a situation that you will work with several applications managed by different teams that use different versions. Let's say the new application uses HL7 v2.4, and if you take a look at CX datatype in https://hapifhir.github.io/hapi-hl7v2/v24/apidocs/index.html, you will realize that it is different from v2.3.1, as shown in Figure 112.

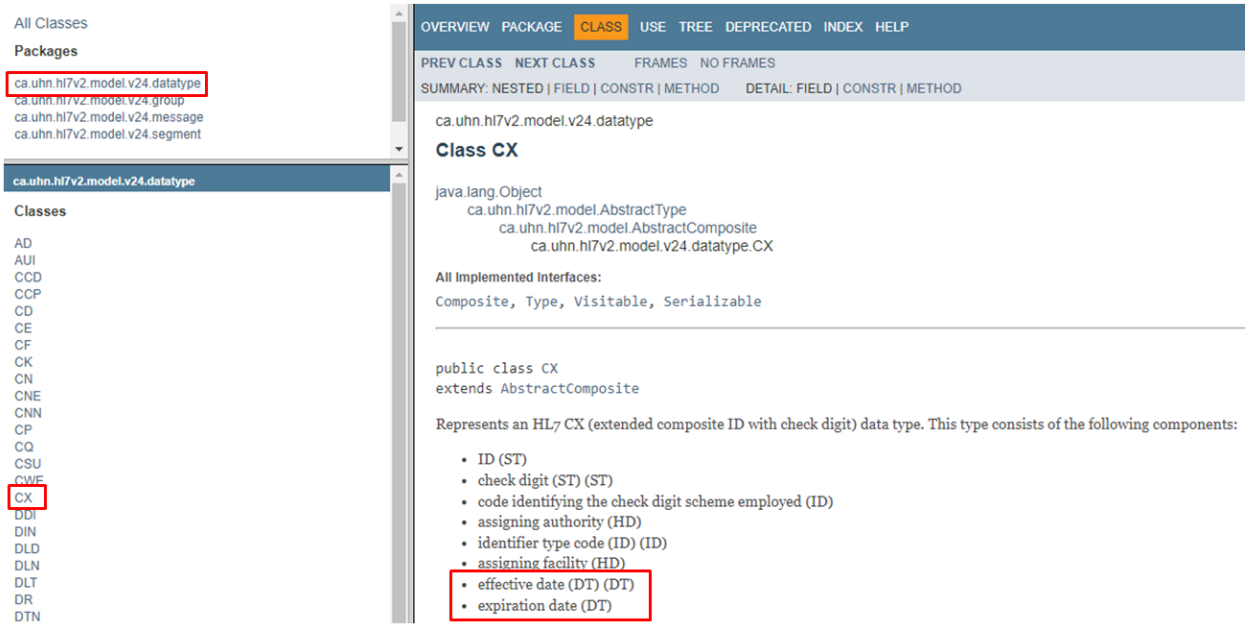


Figure 112.PID-3 components in CX data type (v2.4)

As you see, there are 2 extra components, effective date and expiration date, in the CX data type. This explains that HL7 version was upgraded to 2.4 with this modification, not to mention other segments.

However, if existing old application still uses HL7 v2.3.1 and has to receive ORM message from the new application, there would be a version collision if an old application strictly checks MSH-12 value. Since the new application will auto-populate MSH-12 with 2.4, the old application can't receive the message, and the message will be rejected or filtered. On the contrary, the old application doesn't have a code that checks 7th and 8th components because there is no need to check them in v2.3.1. As a result, PID-3 with 7th and 8th component values will be tolerable. So, in most cases, unless the receiving application doesn't check the specific element, a different format along with an extra value will be accepted.

To allow the v2.4 message to v2.3.1 system, without changing both systems' code, you may work on the message converting project, i.e. populate MSH-12 field with "2.3.1" value and then send the message to an old application. A real-world situation would be more

complicated than this, but this example may give you one of the possible situations that you may face later. In most cases, unless the specific segment, field, component, and subcomponent is strictly checked by the receiving system, mixed format HL7 messages will be allowed. However, the baseline is to follow the HL7 structure according to the specified version in MSH-12.

Subcomponent

Subcomponent is a value separated by "**&**" delimiter in a component. Following PID segment contains subcomponents in red.

`PID|1||1032702^^^SndFac&1.2.3.4.5&ISO^MR^AssignFac&1.2.3.4.5.7&ISO^20190101^20290101||…`

If you look at the Figure 111 or 112, you can see the data type of 4th component, which is HD, as shown below colored in red.

- assigning authority (**HD**)

You can check HAPI site or directly visit (https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/datatype/HD.html) to look at the HD data type, as shown in Figure 113.

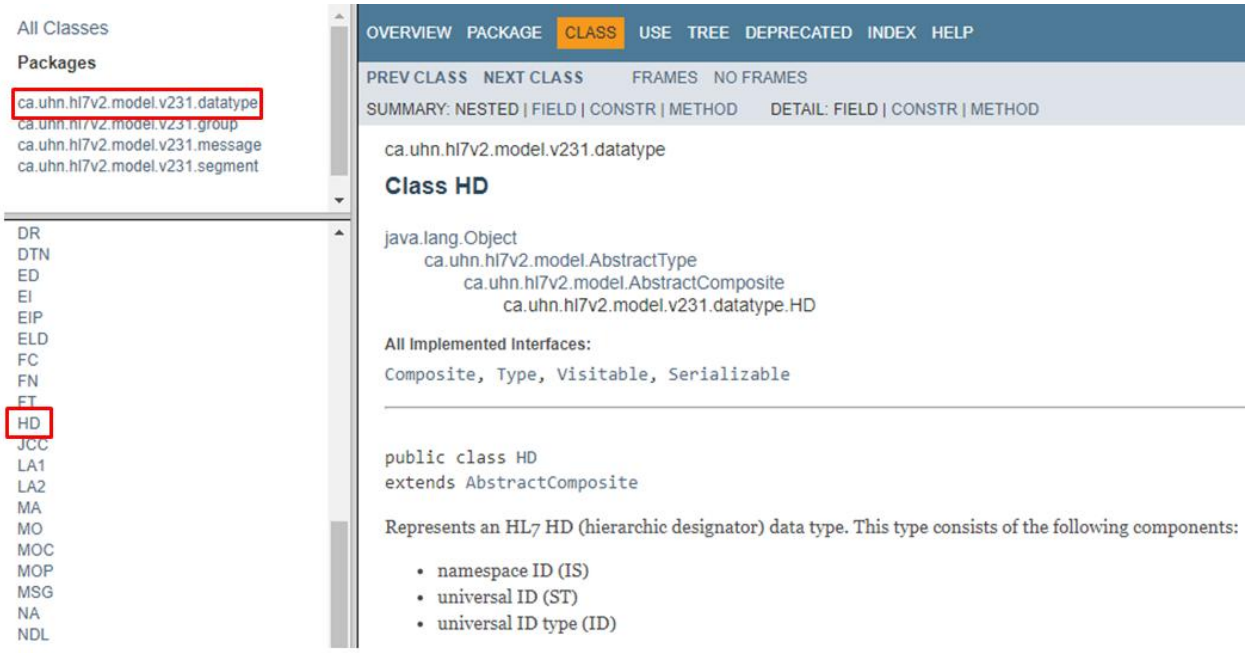


Figure 113.HD datatype for 4th component of PID-3 v2.3.1

And the value of the 4th component of PID-3 could be divided into following Table 16 based on the data type HD.

`PID|1||1032702^^^SndFac&1.2.3.4.5&ISO^MR^AssignFac&1.2.3.4.5.7&ISO^20190101^20290101||…`

Subcomponent ID	Name	Value
1	Namespace ID	SndFac
2	Universal ID	1.2.3.4.5
3	Universal ID types	ISO

Table 16.Subcomponents of 4th component in PID-3(v2.3.1)

HL7 Message exchanging protocol

When transmitting an HL7 message to the destination application through TCP/IP[26], the message needs to be wrapped with MLLP[27] (Minimum Lower Layer Protocol) block. Since the HL7 message is a string of data that is converted to a byte array when transmitted, a receiving application needs to know the start of the message and the end of the message from the received data. Maca's HL7Client

[26] https://www.darpa.mil/about-us/timeline/tcp-ip
[27] https://www.hl7.org/documentcenter/public/wg/inm/mllp_transport_specification.PDF

and HL7Listener provide built-in MLLP handling functions when exchanging HL7 messages through TCP/IP. So, you can focus on composing and parsing messages using embedded HL7-V2 API. We will go over this more in detail when we compose HL7 message later with sample codes. Figure 114 is an implementation of a single HL7 v2 message exchange over MLLP using built-in HL7Client and HL7Listener services. As you see in the ORM Sender, the message is framed with <SB> and <EB><CR> before byte array conversion. And the HL7Listener unwraps the MLLP frame to make the ORM message be parsed using the HL7-V2 API. Although it is a bit early to show this flow, this will give you a brief idea of what we will implement using built-in services.

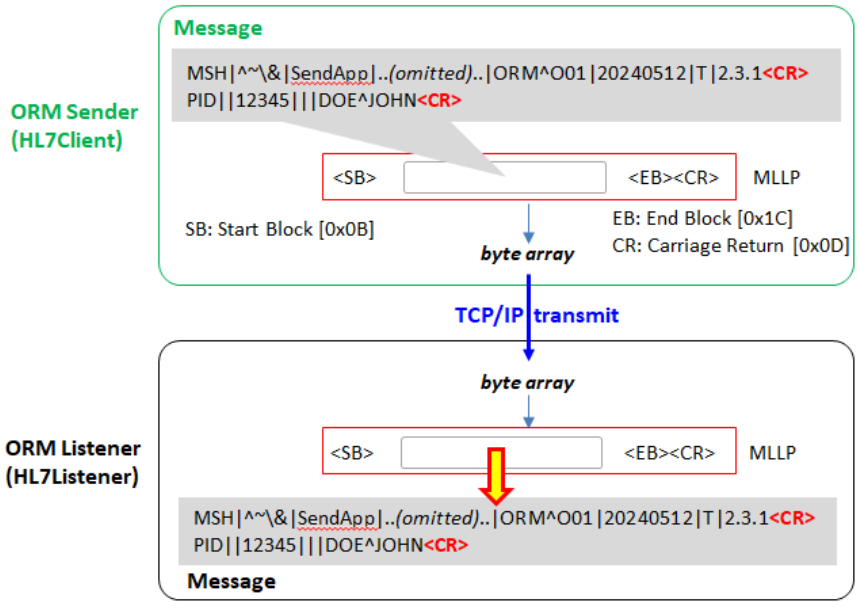


Figure 114.Single message exchange flow (HL7 over MLLP)

Once a receiving application receives an HL7 message, it returns an ACK message[28] right after the receiving or after the defined coded finished with the result value. The ACK message can contain the result of message processing, like Accepted or Rejected. Sometimes, the message sender wants a pre-defined ACK message, such as Received, regardless of the message processing results. We will cover this later with examples.

Compose HL7 message

What we've finished so far in the DICOM Reader service was extracting data from the DICOM files. Now we need to compose HL7 message using those values. Switch to the Development workstation if you are in the Management workstation. Click DICOM Reader from the project explorer on the left. Using service navigator, move to Execution > Steps view.

HL7-V2 library

While we briefly went over the HL7 message, we saw that the HL7-V2 library will be used to parse and compose HL7 messages. If you remember the process that we followed to use fo-dicom library, you may want to find that DLL from NuGet site and download to the project. To support HL7 message handling, Maca is pre-loaded with HL7-V2 library. Once you open DLL Manage popup, you will see the Efferent.HL7.V2 was declared with the using directive, as shown in Figure 115.

[28] https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/message/ACK.html

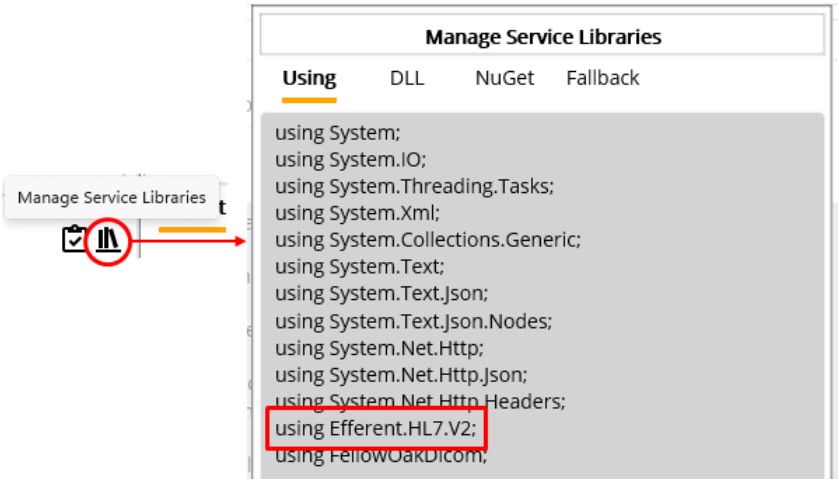


Figure 115.Auto-populated HL7-V2 DLL namespace

For your reference, when you create a new service, the default namespaces including HL7-V2 will be auto-populated. So, you don't have to put the namespace or download DLL for HL7-V2.

Patient Data mapping

As we saw in the Field section above, there are 2 patient ID-related fields; PID-2 and PID-3. For our scenario, we will use PID-2 for the patient ID.

Note: The real-world application may not check the patient ID from the PID-2 if the message is ORM. Always consult with the client about the patient id populating fields.

Following code will extract the patient id and data, then populate them to the corresponding field in a created HL7 message. Since we have working code to parse DICOM, we reused existing code and added HL7 related code, as shown in Figure 116.

As you may already visit, HL7-V2 site provides sample HL7 message creation code in C#, you can visit https://github.com/Efferent-Health/HL7-V2 for the example code.

```
var dicomDir = @"C:\Maca\Dicoms";

StringBuilder buffer = new ();

HL7Encoding encoder = new HL7Encoding();

foreach (var dcm in Directory.GetFiles(dicomDir, "*.dcm"))
{
    var file = await DicomFile.OpenAsync(dcm);

    Message message = new Message();

    Segment pid = new Segment("PID", encoder);

    //Patient ID -2
    pid.AddNewField( file.Dataset.GetString(DicomTag.PatientID), 2);

    //Patient Name - 5
    pid.AddNewField( file.Dataset.GetString(DicomTag.PatientName), 5);

    message.AddNewSegment(pid);

    buffer.AppendLine(message);
}

string hl7File = "HL7"+DateTime.Now.ToString("hhmmss")+".txt";
File.WriteAllText(Path.Combine(dicomDir, hl7File), buffer.ToString());
```

Figure 116.Initial HL7 message creating code without MSH segment

What Figure 116 code does is generate an HL7 message and add PID segment with PID-2 and PID-5 values extracted from DICOM files. And then it saves composed HL7 messages as a text file.

Note: This document is not C# tutorial, and thus will not explain all C# code or related library's code shown here in detail. You can visit Microsoft's C# tutorial site or search web for the C# usage.

At this time, **skip verifying code intentionally**, i.e., do not click the Verify ( ) button. Just click the Save ( ) button and then the Load ( ) button. Once you see the popup, click the Yes button to load the service.

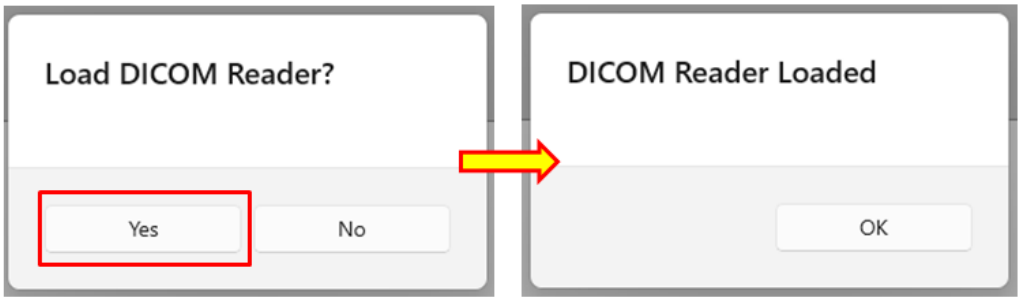


Figure 117.Load modified DICOM Reader service

Switch to the Management workstation. **Click Source tab** if the Instance tab is selected. Click the Refresh ( ) button to show up-to-date source files on the server. Select DICOM Reader if that's not selected, then click the Deploy ( ) button.

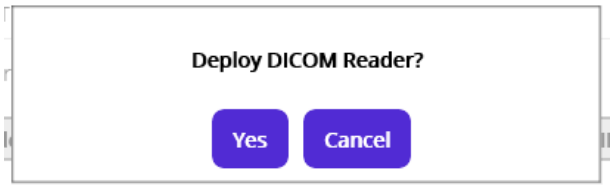


Figure 118.Deploy selected DICOM Reader service

Click the Yes button. At this time, you will see following popup with an error message.

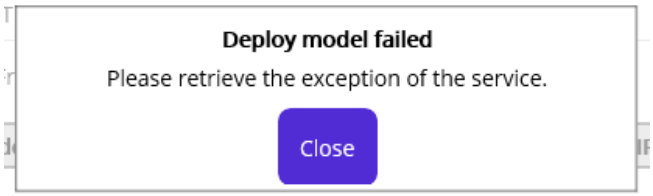


Figure 119.Service deployment error popup

As shown in Figure 119, the service somehow wasn't deployed. To make sure the service is not deployed, click the Instance tab to see the deployed models. Once you click the Instance tab to see the deployed model, you will notice that the DICOM Reader service is not in the deployed model explorer somehow, as shown in Figure 120.

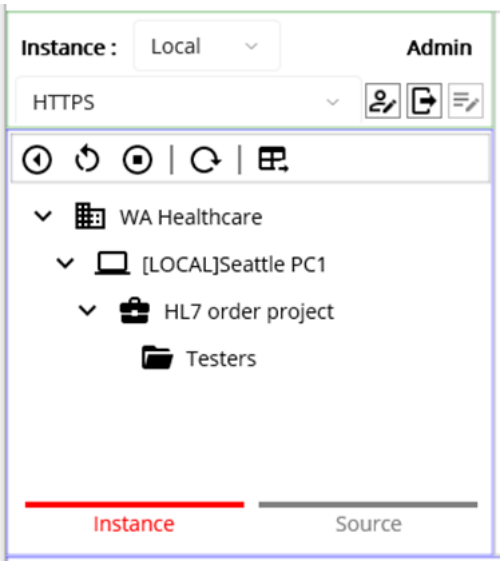


Figure 120.Undeployed DICOM Reader service

Note: The deployment process that we went through earlier does the undeployment process fist, if the service is already deployed. As we didn't undeploy the DICOM Reader service, the service was undeployed first. This is by design. And then the exception happened during the deployment.

As the message recommends, retrieve the exception in the Deploy Exception view that we didn't go over previously. **Click the Deploy Exception tab**. Once you see the view, keep the original options and click the Retrieve button.

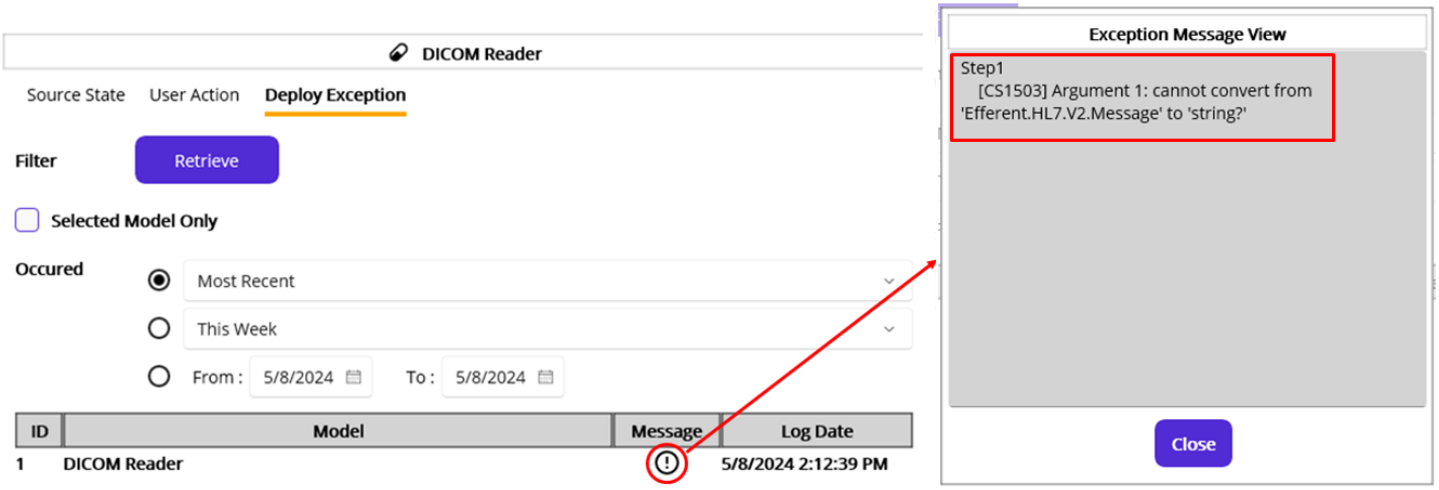


Figure 121.Deploy Exception details

As you see in Figure 121, there is a record for DICOM Reader model. Since this view is a place to retrieve any failed deployment, you can check this view if you don't see the expected Macaron Service in the Instance tab after clicking the Deploy button. In the result table, there is a Message column that has a Message popup (ⓘ) button. Click the button and you will see the detail error message in the popup. As you see, there is a conversion issue from Message to the string in the code that we wrote.

As we did, cross check this by looking at the Source State and User Action views. First, **click the User Action tab** to see the last action. Do not change any options. Click the Retrieve button.

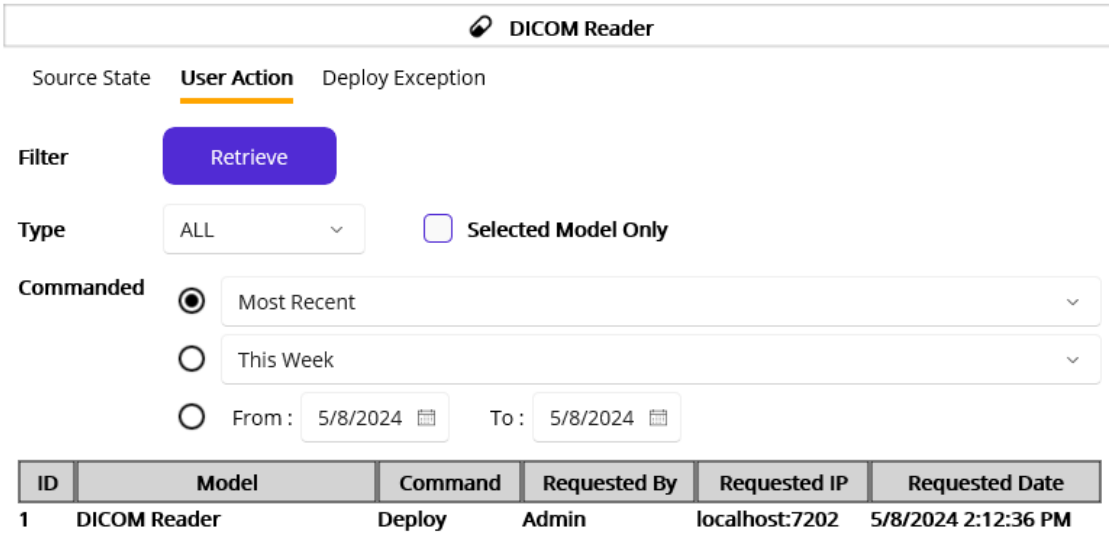


Figure 122.User Action view with the most recent command result

Based on the result as seen in Figure 122, we can confirm that the Admin user initiated the Deploy command for the DICOM Reader. Now, **click the Source State tab** to see the result of the Deploy command. From the options, select Today, which is May 8th, and click the Retrieve button.

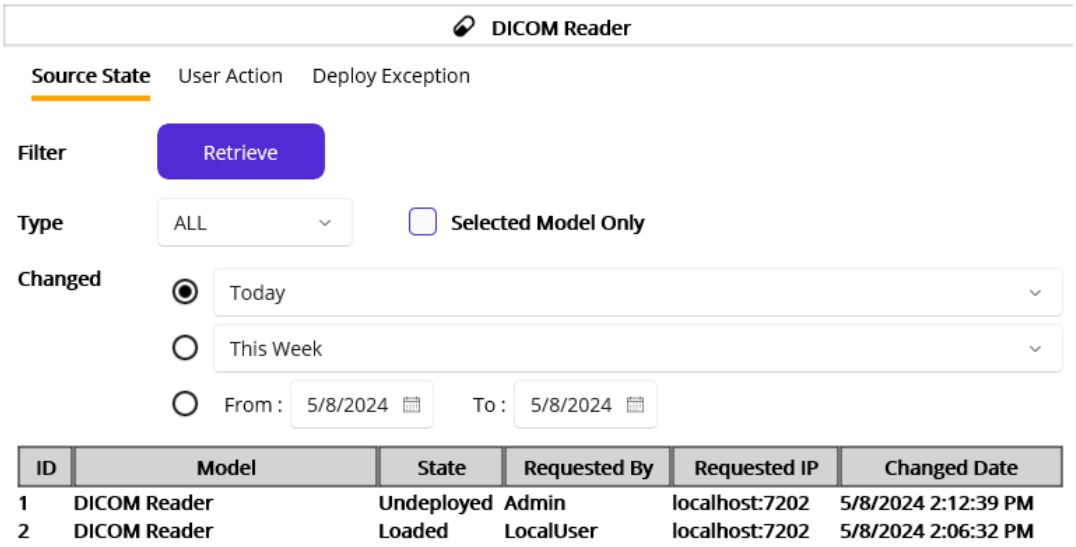


Figure 123.The state history of the DICOM Reader

As seen in Figure 123, there are two records. One is the result of the Load command that we initiated from the Development workstation after the code modification, and the other is the result of the Deploy command initiated by Admin. As mentioned above, the Deploy command checked the deployed model first and then undeployed the model. So, the first record was created as a result. However, as we saw, there was a code error when deploying and the deployment was failed. And no Deployed state record was created. By cross-checking these three views, we can track the services' history that shows what happened.

Note: Although cross-checking gives helpful detail information, still just one view's result can give you handy information. As Figure 123 shows, you can confirm that the DICOM Reader was undeployed and not deployed since the Changed Date if you retrieve it with the Most Recent option. But if you want to verify the Undeployed record is a result of Undeploy or Deploy command, then you can cross-check by looking at the User Action view.

Now we can go back to the editor to debug. **Switch to the Development workstation**. Go to the Execution > Steps view.

As we didn't do the code verification, click the Verify ( ) button to verify modified code to see if there's something that we missed.

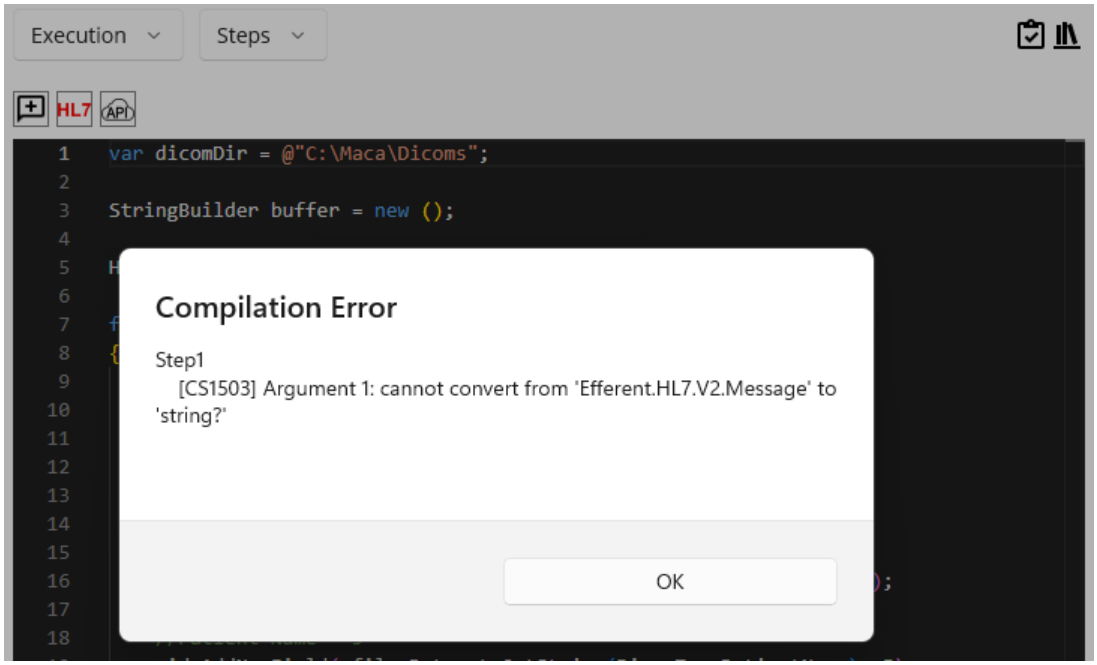


Figure 124.Exception message popup after the code verification

As you see in Figure 124, the exact same exception message is shown. **Please note that in order to show Deploy Exception view example, we intentionally skip the code verification before.** As we did, somebody may deploy a modified service that's not verified. To avoid such situations, **please make sure to verify the code after the code modification**. We may face the error at runtime (execution time), but we have to prevent the code itself from being written incorrectly.

HL7 Message Generator

To facilitate HL7 message composition, Maca provides a helpful UI, HL7 Message Generator. Right above the code editor, click the HL7 ( ) button, as shown in Figure 125.

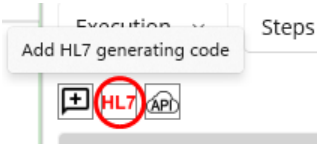


Figure 125.HL7 Message Generator button

Once you click the button, HL7 Message Generator popup shows up, as shown in Figure 126.

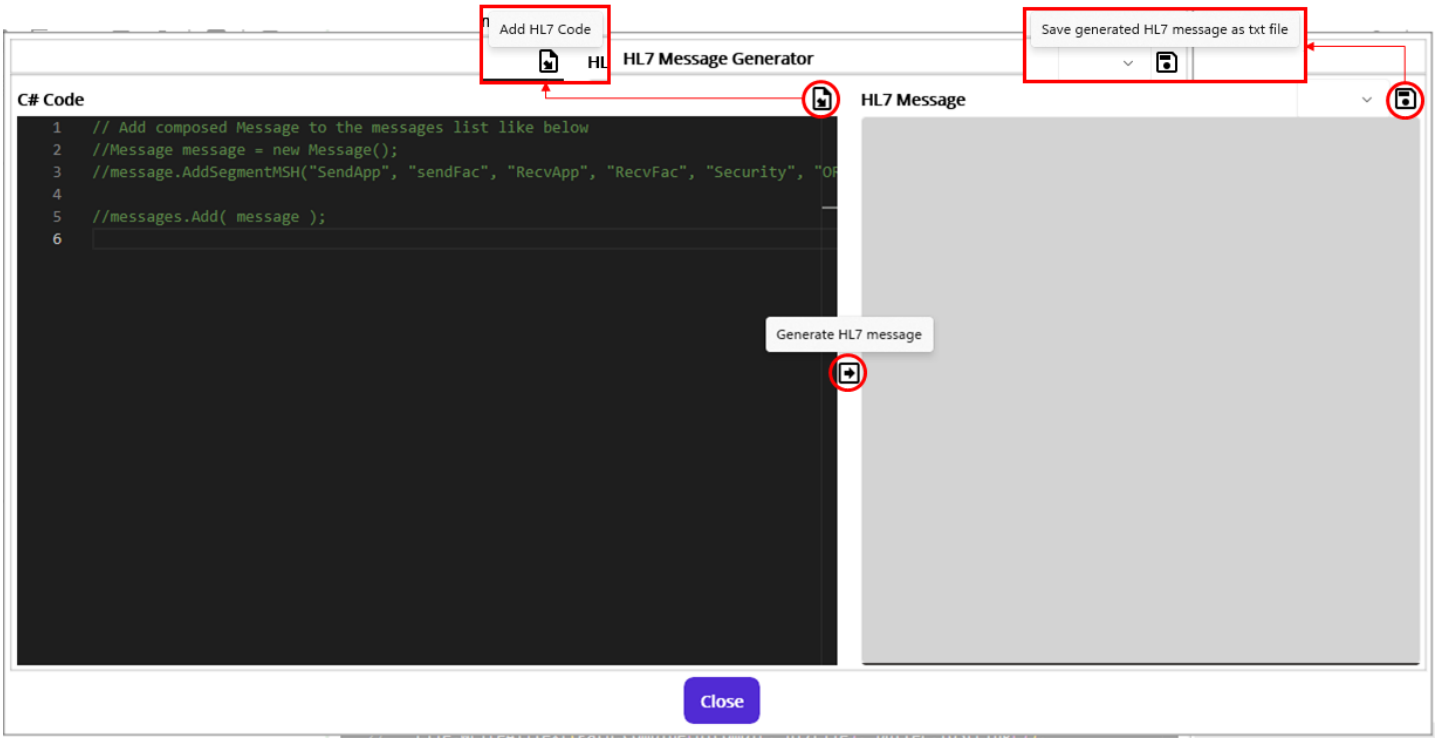


Figure 126.HL7 Message Generator popup with default helpful codes

Using this popup, you can write C# code in the left side editor. Then you can generate HL7 messages by clicking the Generate HL7 message ( ) button in the middle of the popup. Either the C# code is valid or not, the right side shows the result of the code. If you want to save the generated HL7 messages, you can click the save ( ) button. If the code generated the HL7 messages without any error, you can insert written code to the main editor by clicking the Add HL7 Code ( ) button. See below for the detail steps.

As you see at the top of the left editor, as shown in Figure 126, there are four commented lines. You can uncomment the 2nd and 3rd line to generate a message. Then uncomment the last line to add the generated HL7 message to the "**messages**" variable. Please note that the messages variable is a list that holds HL7-V2's Message object. Check below Table 17 for details.

Variable	Type	Usage Example	Detail

messages	**List<Message>**	Message msg = new();
		messages.Add(msg);
		messages.Add(msg.GetACK());
			Only available in the HL7 Message Generator's C# Code editor. Designed to show its element in the HL7 Message display pane, as shown in Figure 127.

Table 17.A messages variable in the HL7 Message Generator

Below code supposed to create an HL7 message with a PID segment and adds to messages.

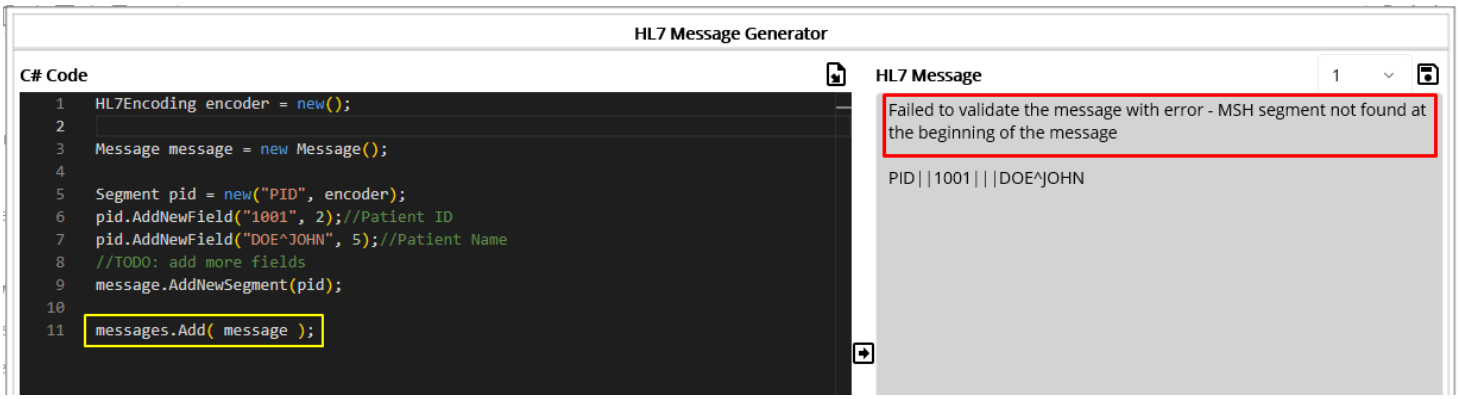


Figure 127.First HL7 message creation code with a missing MSH segment

As you see in Figure 127, especially on the right side pane, there is an error indicates that required MSH segment is not in the generated message. Right below the error message, generated HL7 message is displayed.

As the message directs, add the MSH segment generating code. If you look at the HL7-V2 site, you can find the code does that work, as shown in Figure 128, which is captured from the site.

```
message.AddSegmentMSH(sendingApplication, sendingFacility,
    receivingApplication, receivingFacility,
    security, messageType,
    messageControlId, processingId, version);
```

Figure 128.Adding a message header

As you see Figure 126, you can reuse auto-populated AddSegmentMSH, or you can populate each value by checking HAPI's MSH segment page (https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/segment/MSH.html), as shown in below Figure 129.

Represents an HL7 MSH message segment (MSH - message header segment). This segment has the following fields:

- MSH-1: Field Separator (ST)
- MSH-2: Encoding Characters (ST)
- MSH-3: Sending Application (HD) **optional**
- MSH-4: Sending Facility (HD) **optional**
- MSH-5: Receiving Application (HD) **optional**
- MSH-6: Receiving Facility (HD) **optional**
- MSH-7: Date/Time Of Message (TS) **optional**
- MSH-8: Security (ST) **optional**
- MSH-9: Message Type (MSG)
- MSH-10: Message Control ID (ST)
- MSH-11: Processing ID (PT)
- MSH-12: Version ID (VID)
- MSH-13: Sequence Number (NM) **optional**
- MSH-14: Continuation Pointer (ST) **optional**
- MSH-15: Accept Acknowledgment Type (ID) **optional**
- MSH-16: Application Acknowledgment Type (ID) **optional**
- MSH-17: Country Code (ID) **optional**
- MSH-18: Character Set (ID) **optional repeating**
- MSH-19: Principal Language Of Message (CE) **optional**
- MSH-20: Alternate Character Set Handling Scheme (ID) **optional**

Figure 129.MSH segment's field list (v2.3.1)

Note: Once you open the HL7 Message Generator popup, you will see default commented lines, as shown in Figure 126. Among them you can uncomment AddSegmentMSH line to add MSH segment. Figure 127 shows that the line was intentionally removed. You can replace each field value with another value or just keep the line intact as needed.

Figure 130 shows MSH adding code and the result without errors. The value in the MSH is for demo purpose. More realistic value could be something like "Macaron" for Sending Application field and "EPIC' for receiving application, for example.

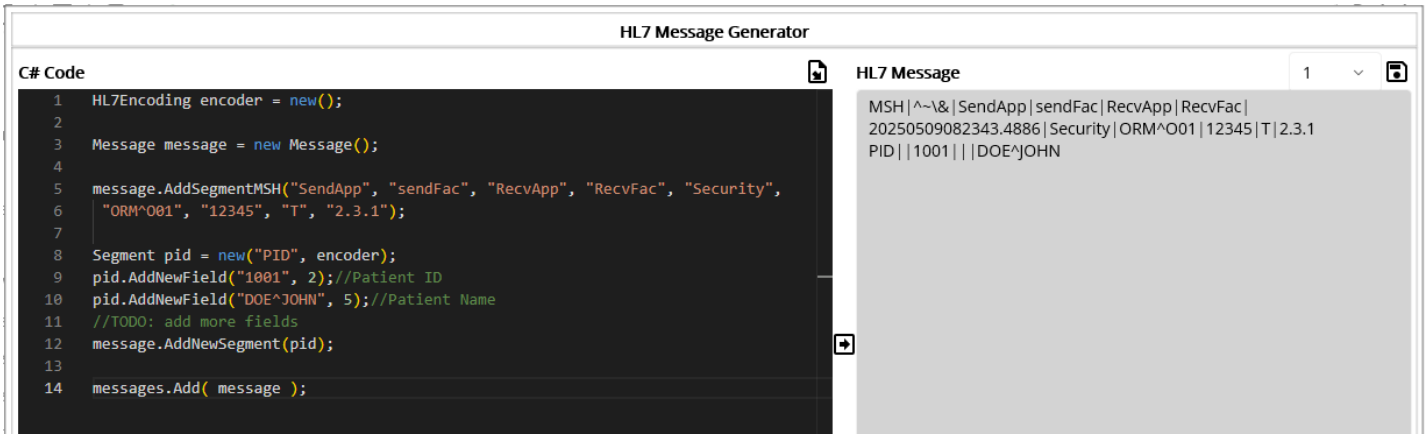


Figure 130.ORM message code

Note: You will work with various applications that populate some fields with system generated values. Of course some fields could be hard coded, i.e., fixed value. But in most cases, you need to set fields based on the running environment, such as Test or Production, or the conditions that communicating application requires.

As we don't have any issue with C# code, let's save the result ORM message as text file. Click the Save ( ) button on the top right.

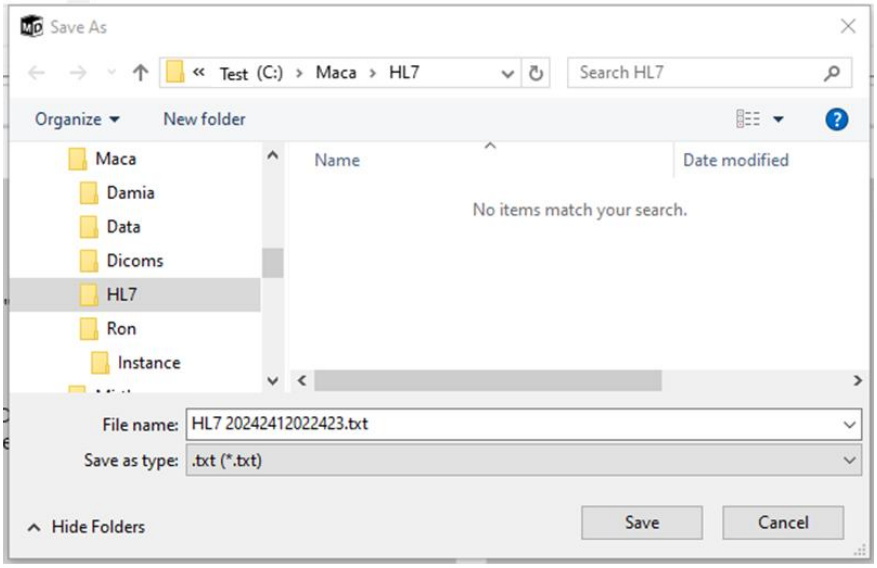


Figure 131.Save generated HL7 message

You may create a separate folder HL7 to store the generated HL7 message. Once you save the file, open it to see the generated ORM message in the text file, as seen in Figure 132.

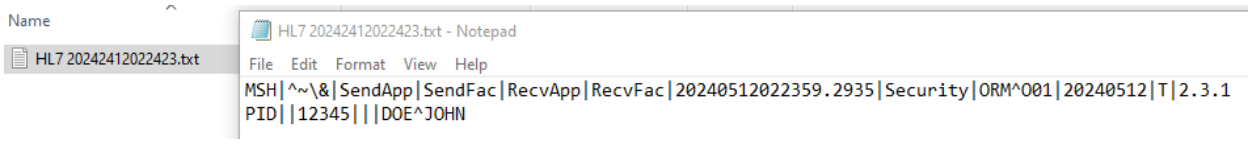


Figure 132.Saved HL7 message

Note: Typically, HL7 messages are stored with an ".hl7" extension. For now, HL7 Generator saves the HL7 message as a text file only. You can change the file extension to ".hl7" manually to use them in the HL7 file processing scenario later. Or, you can create a test service that generates HL7 messages with an ".hl7" extension after adding HL7 code from HL7 Generator popup.

As we checked the generated ORM in Figure 132, let's add C# code that we wrote to the main editor. Click the Add HL7 Code ( ) button. Where the last cursor location, the code we wrote will be inserted. If the HL7 generating code has an error, like Figure 125, the code won't be added and error message popup will be shown, like Figure 133.

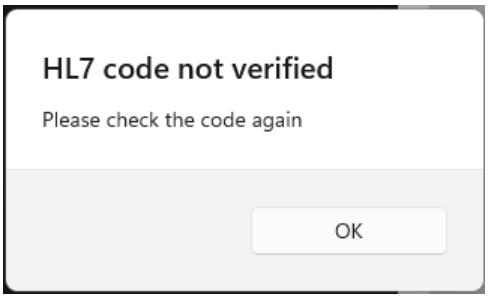


Figure 133.HL7 code not verified popup

As you already noticed, the inserted code compared to Figure 116 is almost the same except for the MSH segment adding code. The purpose of the previous steps was to introduce the HL7 Message Generator. So, please discard any duplicated part and keep the MSH part, as shown in Figure 134. In addition, the previous code that produced an error in Figure 124 was modified as follows.

buffer.AppendLine(message.SerializeMessage(false));

Note: Please check HL7-V2 site for the code example, including **SerializeMessage(false)** method used here.

```
var dicomDir = @"C:\Maca\Dicoms";

StringBuilder buffer = new ();

HL7Encoding encoder = new HL7Encoding();

foreach (var dcm in Directory.GetFiles(dicomDir, "*.dcm"))
{
    var file = await DicomFile.OpenAsync(dcm);

    Message message = new Message();

    message.AddSegmentMSH("SendApp", "SendFac", "RecvApp", "RecvFac", "Security", "ORM^O01", "20240512", "T", "2.3.1");

    Segment pid = new Segment("PID", encoder);

    //Patient ID -2
    pid.AddNewField( file.Dataset.GetString(DicomTag.PatientID), 2);

    //Patient Name - 5
    pid.AddNewField( file.Dataset.GetString(DicomTag.PatientName), 5);

    message.AddNewSegment(pid);

    buffer.AppendLine(message.SerializeMessage(false));
}

string hl7File = "HL7"+DateTime.Now.ToString("hhmmss")+".txt";
File.WriteAllText(Path.Combine(dicomDir, hl7File), buffer.ToString());
```

Figure 134.MSH segment added code

Once modification is done, click the Verify ( ) button to see if the code has any error. If there is no error, load the DICOM Reader service. Then **switch to the Management workstation**. Click the Source tab and click the Refresh ( ) button to show up-to-date files on the server. Select the DICOM Reader service if that's not selected, then click the Deploy ( ) button.

If you see the deployment is done popup, open File Explorer and go to the Dicoms folder where the composed HL7 text file is located. Based on the file name format defined in the code in Figure 134, there will be a text file, as shown in Figure 135.

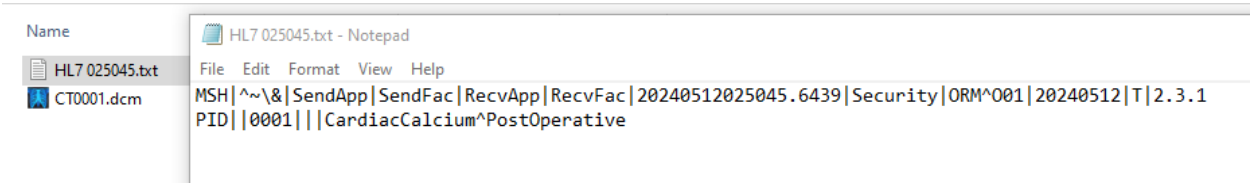


Figure 135.Generated text file containing HL7 message

As we tested in the DICOM parsing section, extracted patient id and name were successfully populated in the ORM message's PID segment. Check Figure 96 for the extracted Patient ID and name values to cross-check.

So far, we extracted the patient information from the DICOM file and populated them to the generated ORM message. Based on the objectives we set, we need to send generated messages to the client application.

Sending and Receiving HL7 messages

As we saw Figure 105, we need two connection-based HL7 message handling ends. A sender could be Macaron and a receiver could be EPIC, or vice versa. Either way, we have to create a standalone service dedicated to sending or receiving the message. If you check Figure 15 again, we have the option to create one of them, i.e. HL7Listener and HL7Client service. For the initial objectives, we need an HL7 message sending service. **Switch to the Development workstation** to build that service.

HL7Client

HL7Client is dedicated to send an HL7 message. This connection-based service requires a target destination, which is IP address and port, to transmit the message. By design, it has an internal queue to hold the HL7 messages to send them to the defined destination, and other services like DICOM Reader must enqueue the message. By separating the HL7 message composing and sending parts in this way, we can focus on the smaller code than the all-in-one code. We will cover how to associate these two services later, with an example.

Create an HL7Client

On the project explorer, select HL7 order project. Then click the Create Project Model button. Once a Create Group or Service popup shows up, select a Service and set values, as shown in Figure 136.

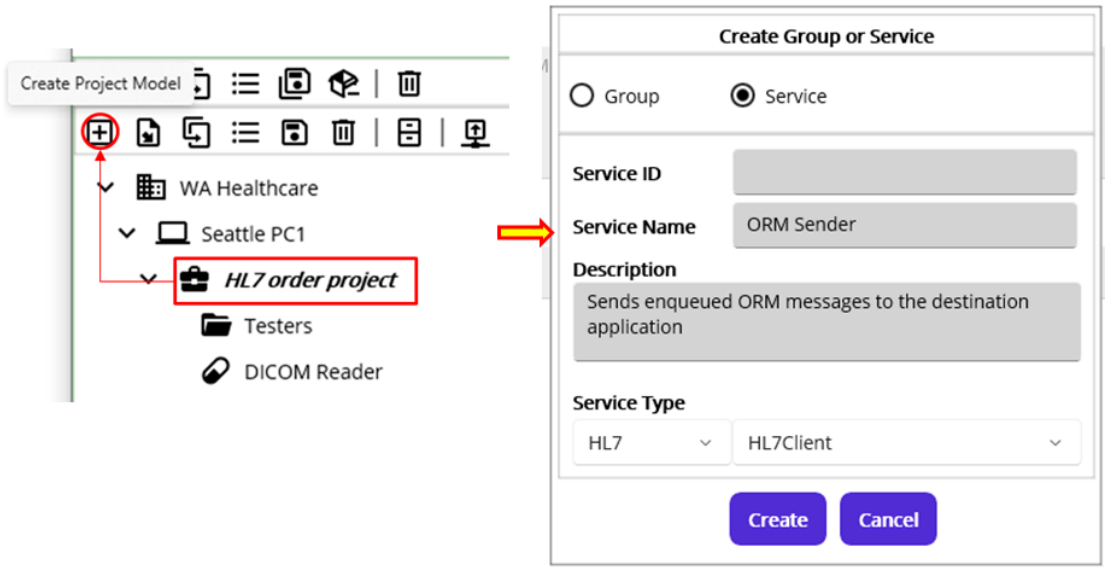


Figure 136.HL7Client service under the project

Once you set all values, click the Create button. Then you can see the service is created under the project, as shown in Figure 137.

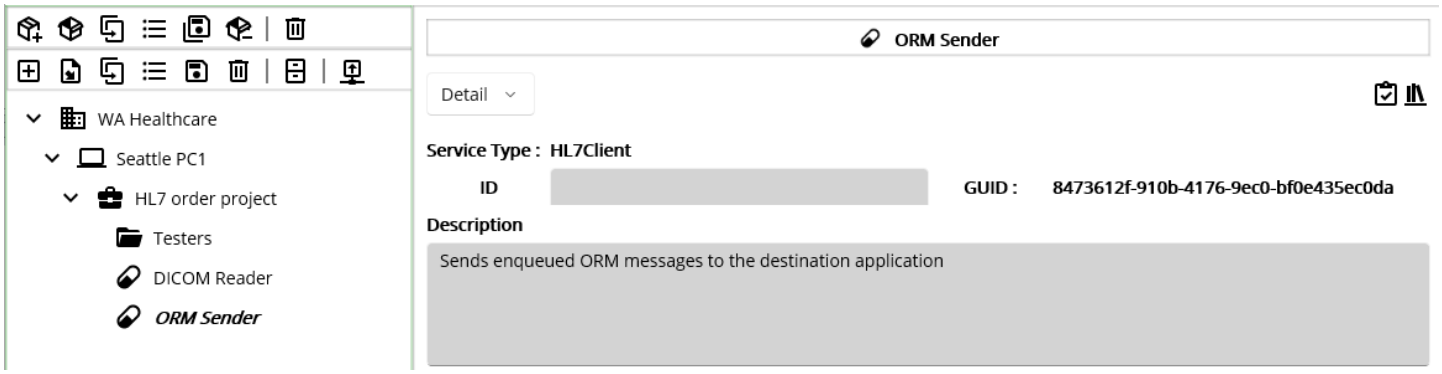


Figure 137.Created HL7Client

Configure the destination

Using the service navigator, go to Settings > Config view. As we need destination information to send an HL7 message, we have to set that in this view. For now, we don't have that information because the client side application is not ready to receive the message. Again, when you start the project with the client, they can't provide the test environment immediately. You have to start building your application with your local environment. So, we have to use a test destination to test this HL7Client, i.e., ORM Sender service. Once your client's test environment is ready, you can replace it with theirs. For now, we can use localhost as the target IP and 8081 as a port to send the HL7 messages. You can set these with your own values if necessary, but for testing purpose, we use this as a basic test address. See below Figure 138.

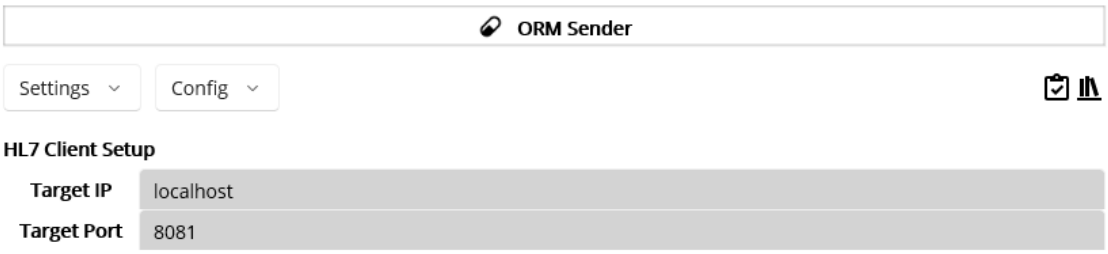


Figure 138.Target destination configuration

Since we set the destination as localhost with port 8081, the destination application should use that information as well. As we will create a test HL7 receiving service later, that service must have the same values.

Note: The HL7Client service requires an active (running) target destination. If the destination is down or the address is incorrect, the service will not be deployed. And thus, the DICOM Reader that enqueues the composed HL7 message will fail as well. We will go through this situation later.

Deploy and Undeploy

As we don't need any specific logic for now, we skip these parts. However, you may put some code if you think that's necessary in Deploy or Undeploy view.

Filter

The main purpose of the Filter view is to check the queued message and decide whether to proceed or discard it. If you trust that the message is always valid, then you don't have to add any code in the Filter editor. However, it is always recommended to send a valid message to prevent the receiving application from validating messages that may cause their code update. So, please make sure to validate message before sending it to them all the time.

Move to Execution > Filter view using service navigator. You will see Filter view with some default commented codes in an editor, as shown in Figure 139.

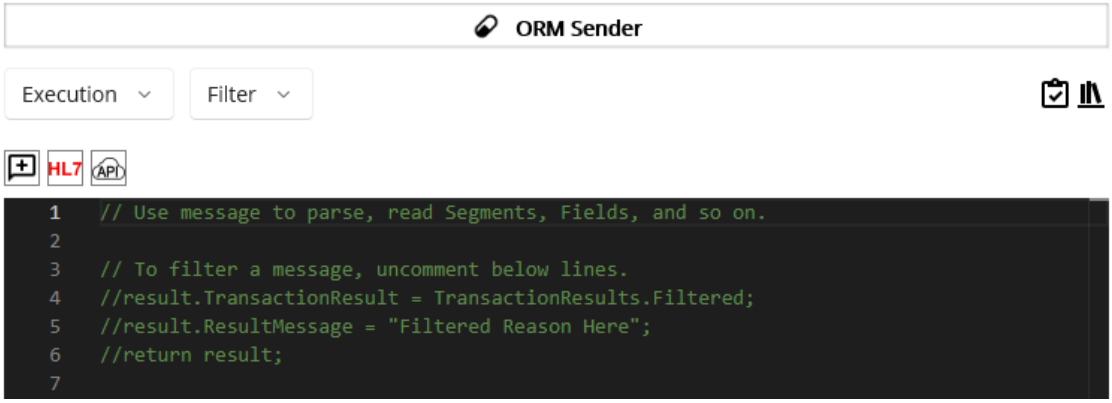


Figure 139.Initial HL7Client Filter View

As the HL7Client has a message queue to send, it is also possible to filter the messages out and not send them. Usually filtering is useful on the message receiving side, e.g., HL7Listener, but the sender can restrict the message as well. In our case, the DICOM Reader service can compose ORM messages without verifying any values, and the ORM Sender service can filter them based on the conditions specified here. Or you don't add any filtering code in the ORM Sender and let the destination application filter what your service sends. You can choose any approach that is appropriate to your situation or based on your preferences. Figure 140 shows possible cases you may consider.

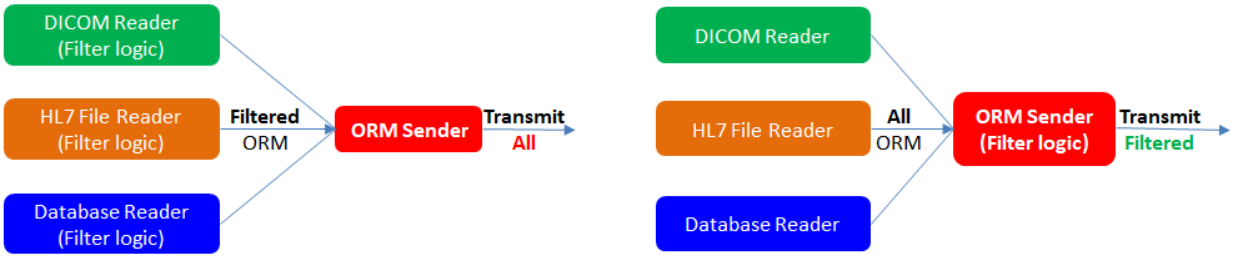


Figure 140.ORM Sender without filtering (Left) and with filtering (Right)

Note: As you see in Figure 140, there is a term "Filtered" in both cases. To support transaction history retrieval, Macaron will keep the message that was **filtered out** in the database as **Filtered**. By design, the message marked as Filtered won't be transmitted. So, the Filtered ORM in the Figure 140 means the message not marked as Filtered. We will cover the records marked with Filtered later.

Based on the DICOM Reader's code, we saw the patient ID is populated to PID-2. What we may filter here is verifying that value, i.e., do not send if there is no value in PID-2. We may not face this situation unless we have sample DICOM files with no patient ID value. So we will intentionally make this case by modifying existing code.

Parse queued Messages

What we saw above is based on the assumption that the queued message has a MSH segment. We should not trust that the received message is valid at all times. In certain rare cases that the DICOM Reader or the other service missed or failed to create an MSH segment, like Figure 116 and 127, and this ORM Sender sends the message anyway, the receiving application may have an issue. With the help of the HL7-V2 API, you can verify whether the generated message, in this case queued messages in ORM Sender, is valid. As we saw Figure 116, there is no problem with the code itself when we verify the code using the Verify ( ) button. But when we send the generated message, this could be a problem. If you do not check MSH and send the message after verifying that the PID has the correct value, the receiving application will not process the message and produce an error as a result. Again, you can use the HL7 Message Generator, as shown in Figure 127 to verify the message beforehand to prevent this situation. Missing MSH segment could be a rare case, but you can handle such a case by using ParseMessage() method if you want, as shown in Figure 131.

Figure 141.Parsing message

This 1 line of the code will parse and **assign internal properties and segments** if there is no error. We will extend the code to show what this means. Since the purpose of the Filter is checking elements of the passed Message and decide proceed or not, internal elements of the message need to be accessible. However, in case of the error like missing MSH segment, this Filter process will stop the rest of the code and throw an error. As a result, Steps code will be skipped. We need an HL7Listener service to demonstrate the issue. But for your reference, **we assume that the HL7Listener service that we will cover later is running**, and the OMR Sender service and DICOM Reader service are running as well. **To follow the below steps, please finish the HL7Listener section first and then come back here.**

Since we have a code to parse queued (received) messages in ORM Sender service, we now set the request mapping. Click the Verify ( ) code in the ORM Sender service.

Request Mapping

Every Macaron service has an Execution section. Once the deployment is done, the service is in a running state. Based on its service type, the service executes the Filter, Steps, and Finalize code. For example, CodeRepeater service executes the code based on the specified time frame repeatedly, as the service name itself describes. So, there is no need to trigger the service to execute the code from the outside of the service. On the contrary, HL7Client does not execute code automatically while it's running; the other service has to request the HL7Client service to execute the code by passing a message in the request. Lastly, the HL7Listener that we will create later is dedicated to communicate with the HL7Client service or external application, such as Mirth. Once the message transmitted from the HL7Client or external application has come, the service will execute the specified code with that message. So, the HL7Listener service can't be directly requested.

Note: The current prototype version doesn't have a RequestListener service that could be requested by all other services. The service will be added in the next release.

Select the DICOM Reader node in the project explorer. Go to **Execution > Steps** view using service navigator. **Click the last line** of the foreach block. Then click the Service Request ( ) button, as shown in Figure 142.

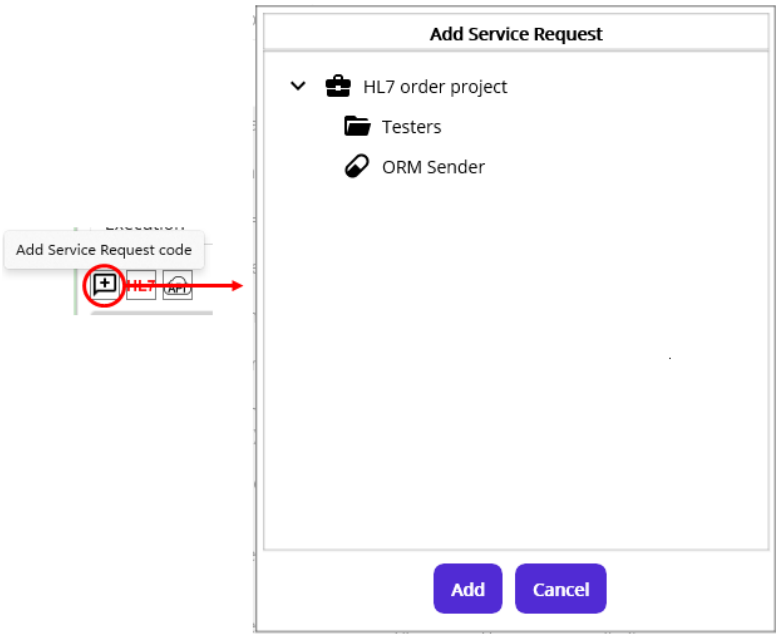


Figure 142.Add Message Request popup

As you see, only ORM Sender is visible even if we have DICOM Reader service as described above. Please note that what the Add Message Request popup does is show requestable services and map to send a message from the current service. Since we are in the DICOM Reader service, specifically in the Steps code, we can request the ORM Sender service to execute its code with a message in the request. Select ORM Sender service and click the Add button. Once the popup disappears, you will see inserted code, as shown in Figure 143.

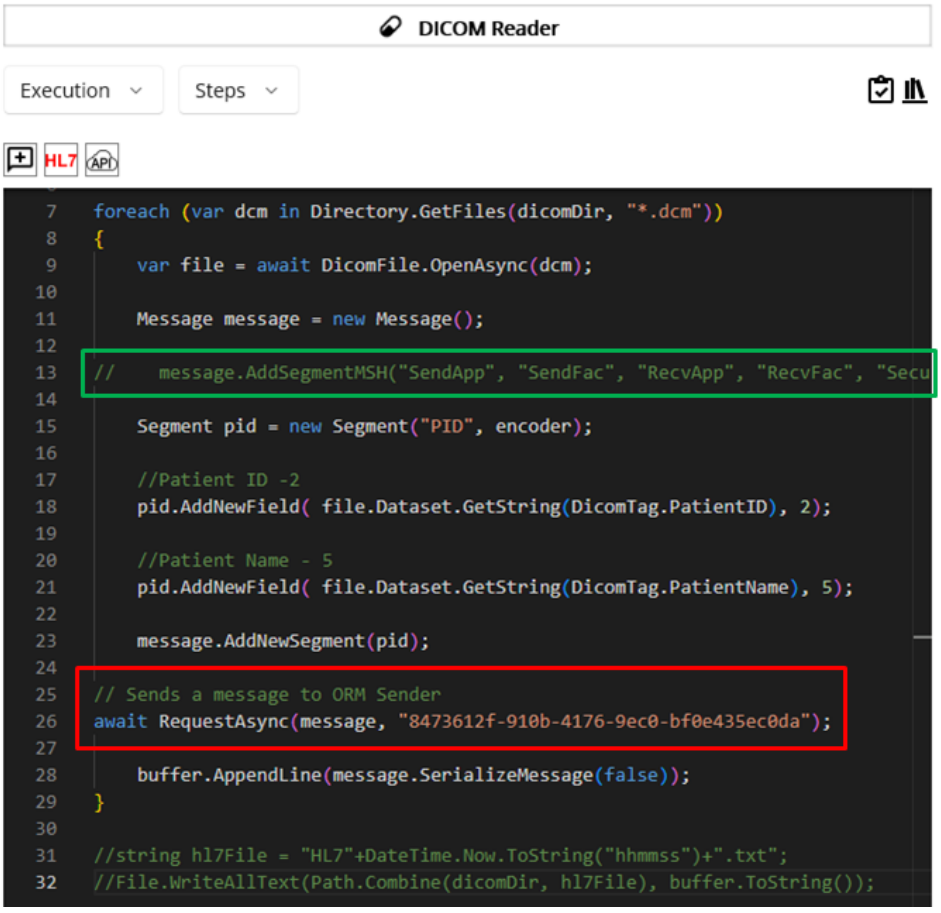


Figure 143.Inserted Message Request code and commented MSH line

Inserted code is Maca's awaitable[29] Request method that requires a string value of message to send and the target service's GUID. You can use as is, but once you use a different variable name, like "data" or "msg," instead of "message" in the editor, change "message" variable inside Request method accordingly. In addition, you can keep existing data saving code or remove. Following Table 18 shows the argument list and its usage example when you use overloaded Request method. As you see, you can pass either string value or HL7-V2's Message object in the 1st argument location.

1st argument	2nd argument	Example
string value of HL7 message	GUID of the target service (auto populated)	**string** hl7 ="MSH|^~\&|SndApp…"; await Request(hl7, "8473…");
Message object	GUID of the target service (auto populated)	**Message** hl7 = new ("MSH|^~\&|SndApp…"); await Request(hl7, "8473…");

Table 18.Maca's Request Method

Along with that, to test the Filter section of ORM Sender, as shown in Figure 143, MSH segment creating code was commented.

Save both DICOM Reader and ORM Sender services. Then load both services. Once the load is done, **switch to the Management workstation**. Click the Source tab and click the Refresh button to show up-to-date files on the server. Select ORM Sender service and then click the Deploy button. Once the deployment is done, deploy DICOM Reader service. Then click Observe > Transaction view. As DICOM Reader service is selected, click the Retrieve button to see the latest transaction record, as shown in Figure 144.

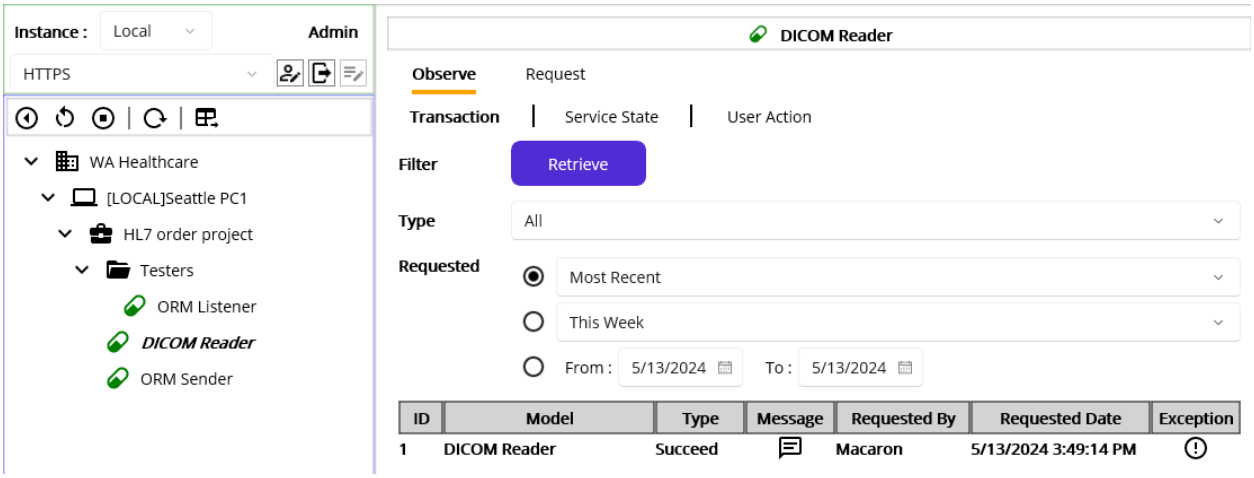


Figure 144.DICOM Reader's most recent transaction data

As the result shows, we had no issue to process the code. Now, select ORM Sender and click the Retrieve button. You will see a record with the Error type, as shown in Figure 145.

[29] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await

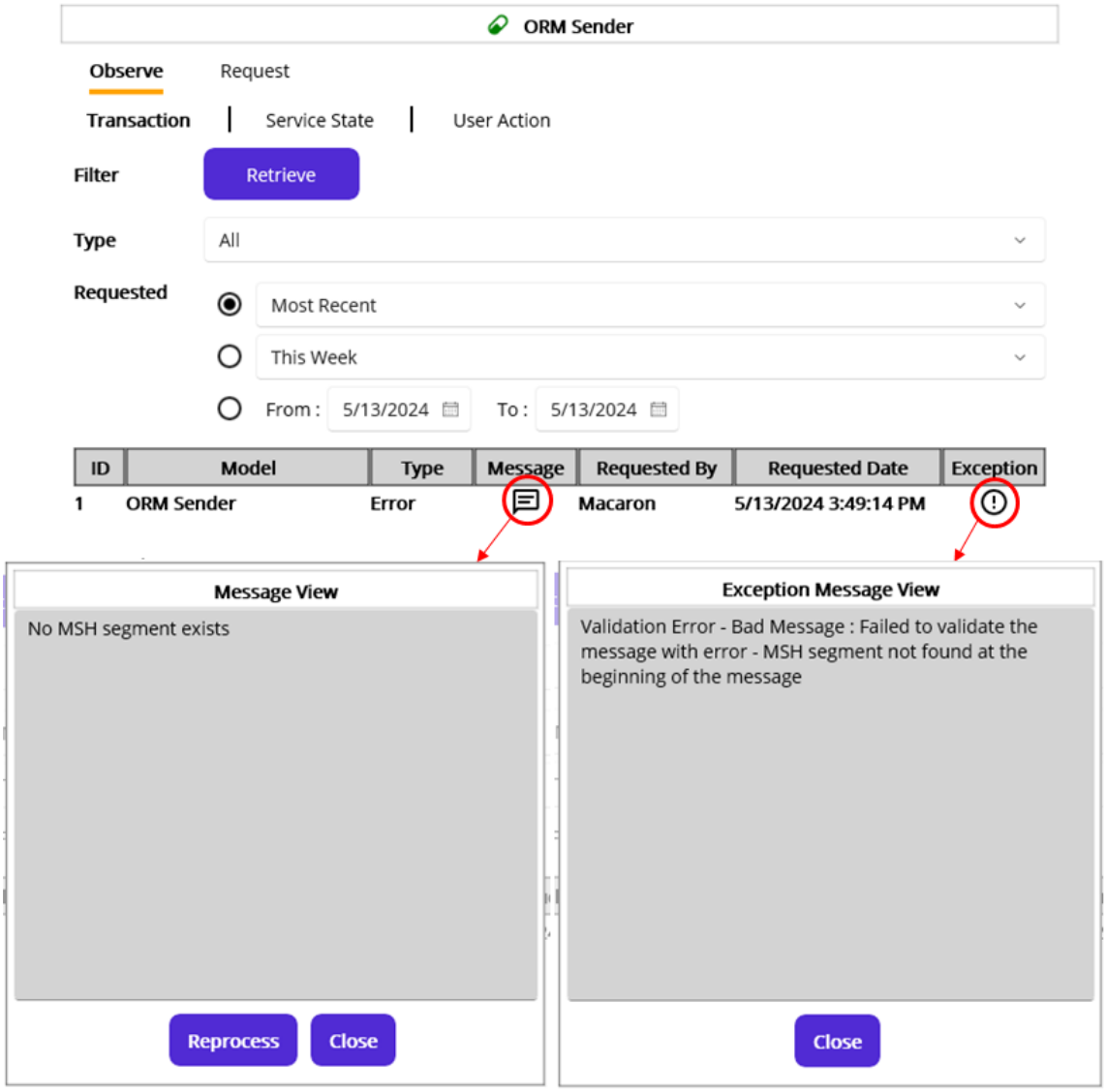


Figure 145.A transaction with an error

Once you see the result, click both Message and Exception buttons to see the detail. Since we sent a message without MSH, Message View popup shows “No MSH segment exists” fallback message. And the Exception Message View shows detail about the error occurred in the Filter code. Please note that **ParseMessage()** validates the message internally, **sets all properties and segments**, and throws an exception if the message is not valid. And the Filter view internally catches the exception and sets the transaction as an Error. Then the error message was saved to the database.

Alternatively, if you want to mark the exception thrown by ParseMessage() as Filtered instead of Error, you can catch the exception inside the Filter and return the result as Filtered, as shown in the below Figure 146. We will cover the result variable shortly.

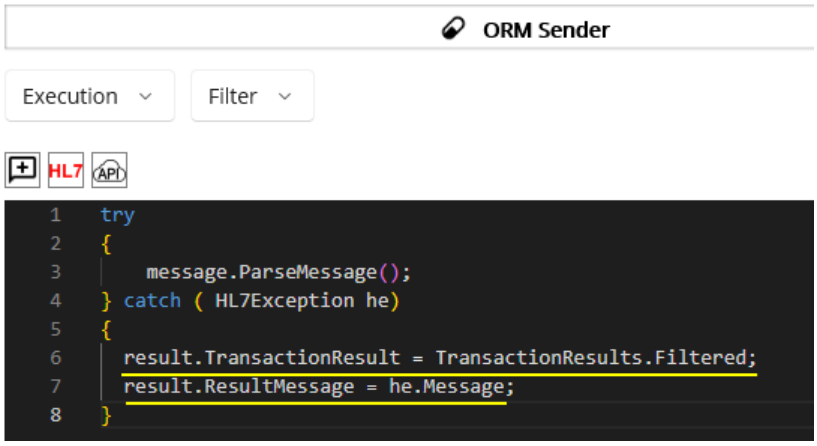


Figure 146.Catches an error in the Filter

If you reload and deploy updated ORM Sender, you will see the transaction was filtered, as shown in Figure 147.

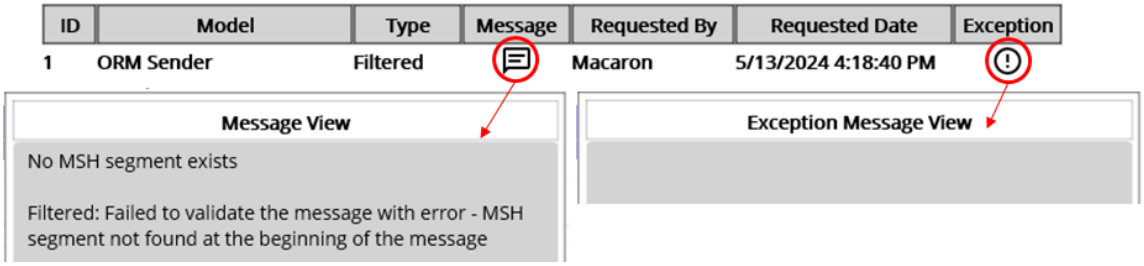


Figure 147.Filtered message

But at this time, you will not see any message in the Exception View since we set the result as Filtered. As we set an exception message to the result.ResultMessage property in Figure 146, the caught exception's details will be in the Message section. For now, the Exception column will have a message only when an error occurs. You can keep this code, but for this tutorial, we do not wrap ParseMessage method with try catch block. **Switch back to Development workstation** and select ORM Sender node from the project explorer to continue filtering.

Message filtering

Once the message parsing is finished without an error, we can add filtering code. **Uncomment MSH line** in Figure 143. As briefly mentioned, we can check a certain field's value to filter the message using ***message*** variable. For example, we can restrict ORM Sender to accept only ORM message, as shown in Figure 148.

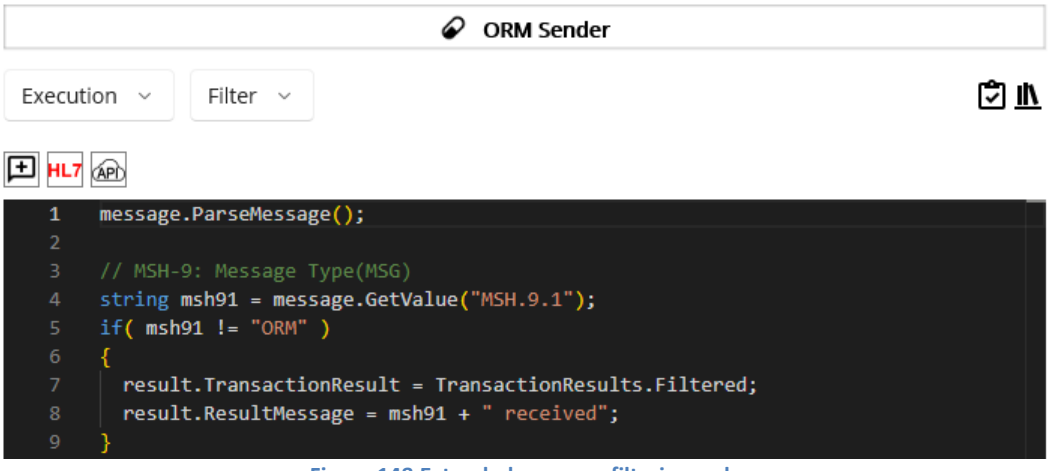


Figure 148.Extended message filtering code

As you see in Figure 148, you can use the pre-defined "***message***" variable, which is HL7-V2's Message object, to access the dequeued HL7 message. With that variable, you check the values in a Segment, Field, Component, and Subcomponent. Table 19 shows available pre-defined message and result variables that you can use in the Filter view.

Variables	Type	Usage Example	Detail
message	Message	message.ParseMessage();	HL7-V2's object
result	RonResult	result.TransactionResult = TransactionResults.Error; result.ResultMessage = "Error message here"; return result;	Maca's object

Table 19. message and result variables

Please note that the code checks the MSH-9's first component and then sets Filtered enum value to the pre-defined "***result***" object's TransactionResult property if the message is not ORM. **For now, "*message*" and "*result*" will be visible for both Filter and Steps views.** So you can use them to access and set as needed. See below Table 20 for RonResult's property list.

Property Names	Type	Detail
TransactionResult	TransactionResults **enum**	Result of the transaction
FailedStepId	**int**	Maca will populate. For now always 1. This is internal use only.
ResultMessage	**string**	Assign it with the message if necessary. In case of unhandled exception, Maca will populate it with Exception.Message.

Table 20.Maca's RonResult result object's properties

And below are Maca's TransactionResults enum values that you can set as the result's TransactionResult property.

enum TransactionResults { Succeed, Error, Filtered }

Note: Since ParseMessage() verifies required fields and assigns all segments and properties using HL7 Message property, without calling it, accessing an element of the dequeued Message will fail, as shown in Figure 149.

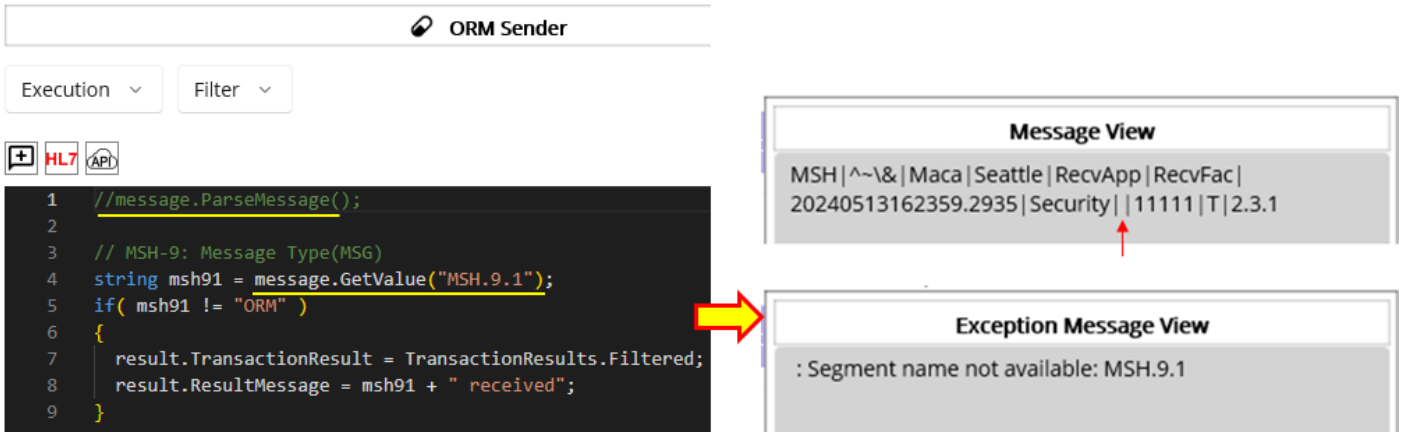


Figure 149.Unassigned segment accessing error

By Maca's design, a Message in the HL7Client's queue will have an HL7Message property only. All other segments and properties will be empty. As mentioned above, ParseMessage() will check the required segments and fields and throw an exception if any of them is missing, and Maca will catch that. If the message is valid, then all segments and fields will be populated based on the HL7Message property. But if we comment that, the subsequent code that tries to check a segment or field will fail because that is empty. If an invalid message was queued, e.g., required field MSH.9.1 is missing, the code that checks MSH.9.1 will throw an exception and Maca cacthes that and save that in the database. Then you can retrieve the exception message in the service's transactions, as shown in Figure 149. Although we will not face this situation because the DICOM Reader's "AddSegmentMSH()" validates the header value and throws an exception if the required field is missing. Then no message will be queued to the ORM Sender as a result. But we still need to test the ORM Sender with various messages without using the DICOM Reader. We will cover that later.

Lastly, once you use “return result;” code inside the Filter, the rest of the code will not be executed followed by C# language standard. Plus, as we filtered the message, Steps code that we will define later will not be executed as well.

Note: Maca provides its own internal classes and enums along with extension methods for the 3rd party libraries. Since the prototype doesn’t provide fully functional code completion, e.g., Microsoft’s IntelliSense[30], in the code editor, please check each one’s specification stated in the tutorial carefully.

To test the MSH-9 field verification, i.e., to accept only ORM message, modify DICOM Reader service, as shown in Figure 150.

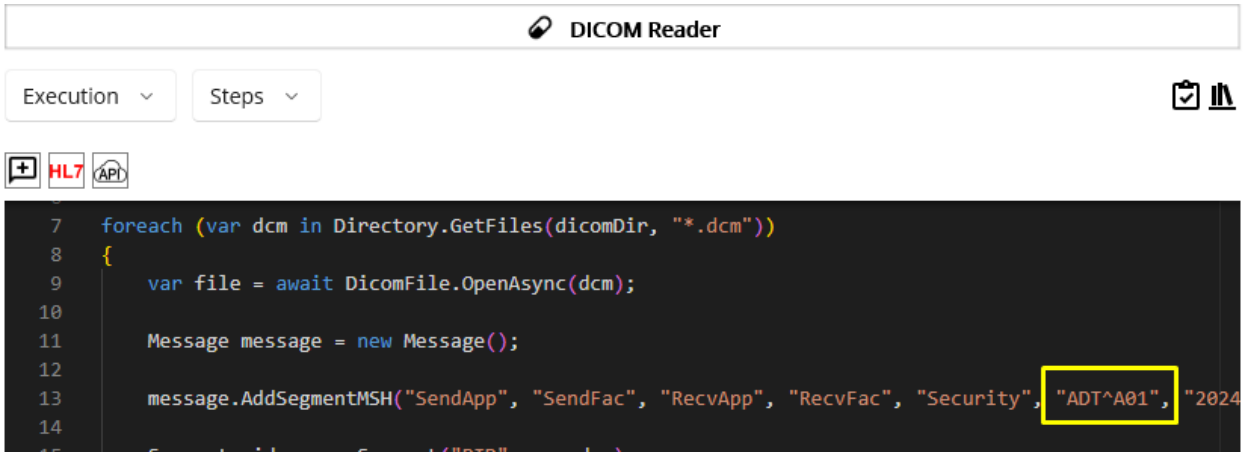


Figure 150.ADT message example

Note: Below steps are for your reference. The ORM Listener service needs to be deployed first. Once you finish below HL7Listener section, come back here to follow the steps.

Save the DICOM Reader and ORM Sender services and reload them. If the load is done, **switch to the Management workstation** and click Source tab. Click the Refresh button. Select ORM Sender and click the Deploy ( ) button. Then select DICOM Reader and click the Deploy ( ) button. Once the deployment is done, click Instance tab. Select ORM Sender and then click **Observe > Transaction** tab if that is not selected. Click the Retrieve button and you will see the most recent result, as shown in Figure 151.

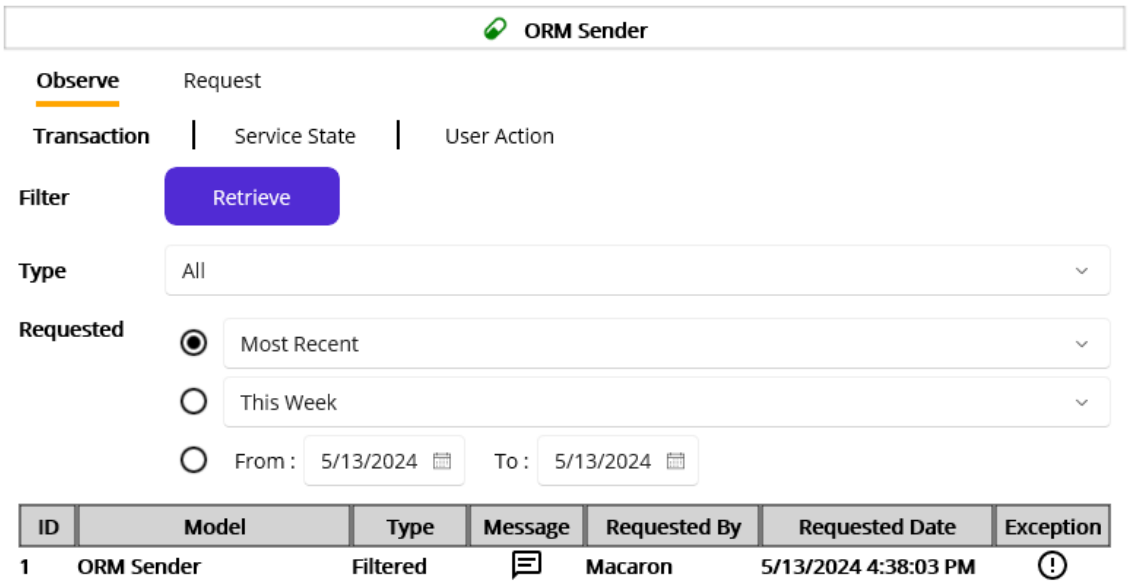


Figure 151.Filtered message

[30] https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2022

At this time, the message was filtered due to the MSH-9 value, which is ADT^A01 in this case. As this message contains MSH segment, we may look at the Message view. Click the Message ( ) button.

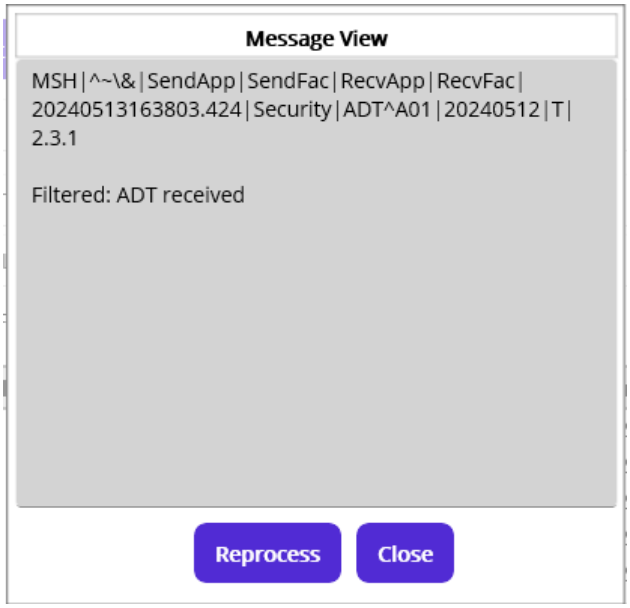


Figure 152.Received (*queued*) HL7 message

Figure 152 shows the received message. As you see, you will notice that the message was filtered because the MSH-9 is ADT^A01.

This is minor, but one thing to note is the components of the MSH-9, which is for the message type. As you saw, the field could have all components like "ORM^O01^ORM_O01" in the above sample MSH. But in many cases, you will see the value like "ORM^O01" in MSH-9. So, parsing a field value without checking its components may lead to an incorrect result. In addition, if you try to check the 3rd component of the MSH-9, i.e., MSH.9.3, you won't have the expected outcome even though the message is valid ORM. So, please make sure to check each component first, once you try to check the field.

Like the above test, convert ADT^A01 back to ORM^O01 and modify MSH-12 to 2.4 in the DICOM Reader. Then deploy the services to see whether the transaction was filtered.

As an extra example, we can check if the required segment exists. Let's say we need a PV1 segment in every message and do not proceed if the segment doesn't exist. For now, the DICOM Reader creates MSH and PID segments.

```
if( message.Segments("PV1").Count ==0 )
{
    result.TransactionResult = TransactionResults.Filtered;
    result.ResultMessage = "Missing PV1 Segment";
    return result;
}
```

Figure 153.Filter the message if PV1 segment does not exist

Since the segment can repeat, as mentioned in the Segment section above with an OBX example, HL7-V2 API always checks the segment in a plural way, like Segments() method. So, you can check the segment using the Count property, which returns the number of the existing segment. You can modify ORM Sender and redeploy it to see the result.

Note: Although you can set filtering code here, it is still possible to add filtering code in the Steps view. Since the *result* variable will be visible in the Steps view, you can use the same filtering code there to do not send a message by using the *result* variable. However, if you want to make the code clear, put the filtering code only in the Filter view.

HL7 data security

As you will handle HL7 messages, it is quite important to protect the sensitive data inside the message. Since the Macaron can be installed on mobile devices, such as laptops, it is open to theft if it is not attended to. If you store the entire message on the portable device in some way, i.e., either encrypted or as plain text, personal data in that file could be the target of malicious activities. Please note that most HL7 messages contain administrative or demographic data, such as address, phone number, and SSN, along with the medical history, not to mention the insurance. If that data is disclosed, it will be a violation of HIPAA[31] compliance. So, **do not save HL7 messages to the device where the Macaron was installed at any time**. As we did before, we can save an HL7 message using HL7 Message Generator. But, that should be for testing and debugging purpose only. Sometimes the other application may save HL7 files to be processed later, but the saving location could be in the remote server protected by username and password. For this reason, Macaron does not save the entire HL7 message of each transaction to the embedded database. Instead, to facilitate the transaction retrieval and monitoring, the header data of the message will be saved, as shown in Figure 145 and 152. However, there would be a situation where you need to reprocess the transmitted message, which was valid HL7 message but failed due to some reason. And the client system may not have the capability to resend specific previous message. To cover that situation, we may need a separate location to save the received messages and pick a message to reprocess. For now, the prototype version doesn't save the entire message by design, and thus can't reprocess previously received message. But in the later release, this feature could be added with a design modification.

Steps of code before send

You can send the message as is to the destination if you are sure that the queued message is fine, or if the destination side handles the situation. But in some cases, like when the DICOM Reader service can't finalize the message due to some reason, it might be required to modify the message here before sending the message to the destination due to the following possible reasons:

- Move a repeating code in the CodeRepeater services to the HL7Client, like Figure140.
- Certain element, such as a field or component, can be modified or populated in the HL7Client.
- Add a new segment that is more relevant in here.

There will be more cases that we may think of, but the above will give you some idea of what needs to be considered in the HL7Client's Steps view.

As we extracted patient id and name from the DICOM file, we may use that information to extend the code in the ORM Sender. As you saw the code in Figure 143, we put the extracted patient id to the PID-2 field.

But at this time, we can think about the first-time walk-in patient case. Since the patient was not in the system, there is no system generated patient ID assigned to the visitor. And the DICOM generating device doesn't have a direct connection to the patient system, such as EMR or EHR[32]. But the device provides a patient ID input field along with all demographic data input fields, including name, and saves that information internally. A doctor or nurse types all the information, except the patient ID. Then the DICOM file will be exported with that patient's information. Since there is no patient ID in a DICOM file, the ORM we generated doesn't have the patient ID as well. Unfortunately, ORM receiving application doesn't register a patient ID if PID-2 is missing, and thus rejects the message. So, we must obtain a new patient ID by calling the patient registration service if PID-2 is missing. We will implement the ORM receiving application using HL7Listener later. Although the case explanation was quite verbose, we will not implement the external service call here. Since this is to guide the usage of the Steps view, we will assume we called that service.

Note: As mentioned, this is a hypothetical situation. A real-world case would be different than this, e.g., DICOM generating device may call the patient registration service directly. And the ORM receiving application can register the visitor and generate a patient ID if no PID-2 is populated in the received ORM message.

Go to Execution > Steps view. Similar to Filter, it shows the default comment about the message variable, as shown in Figure 154.

[31] https://www.cdc.gov/phlp/publications/topic/hipaa.html
[32] https://www.healthit.gov/faq/what-electronic-health-record-ehr

Figure 154.Initial Steps view

As the comment directs, you can use the same code, i.e., call message.ParseMessage() to access the message's elements. Below code is an implementation of the above hypothetical case. **If you used ParseMessage() method in the Filter editor, there is no need to call the method again here because the elements are ready to read.** Since we used in the Filter, we comment this at this time.

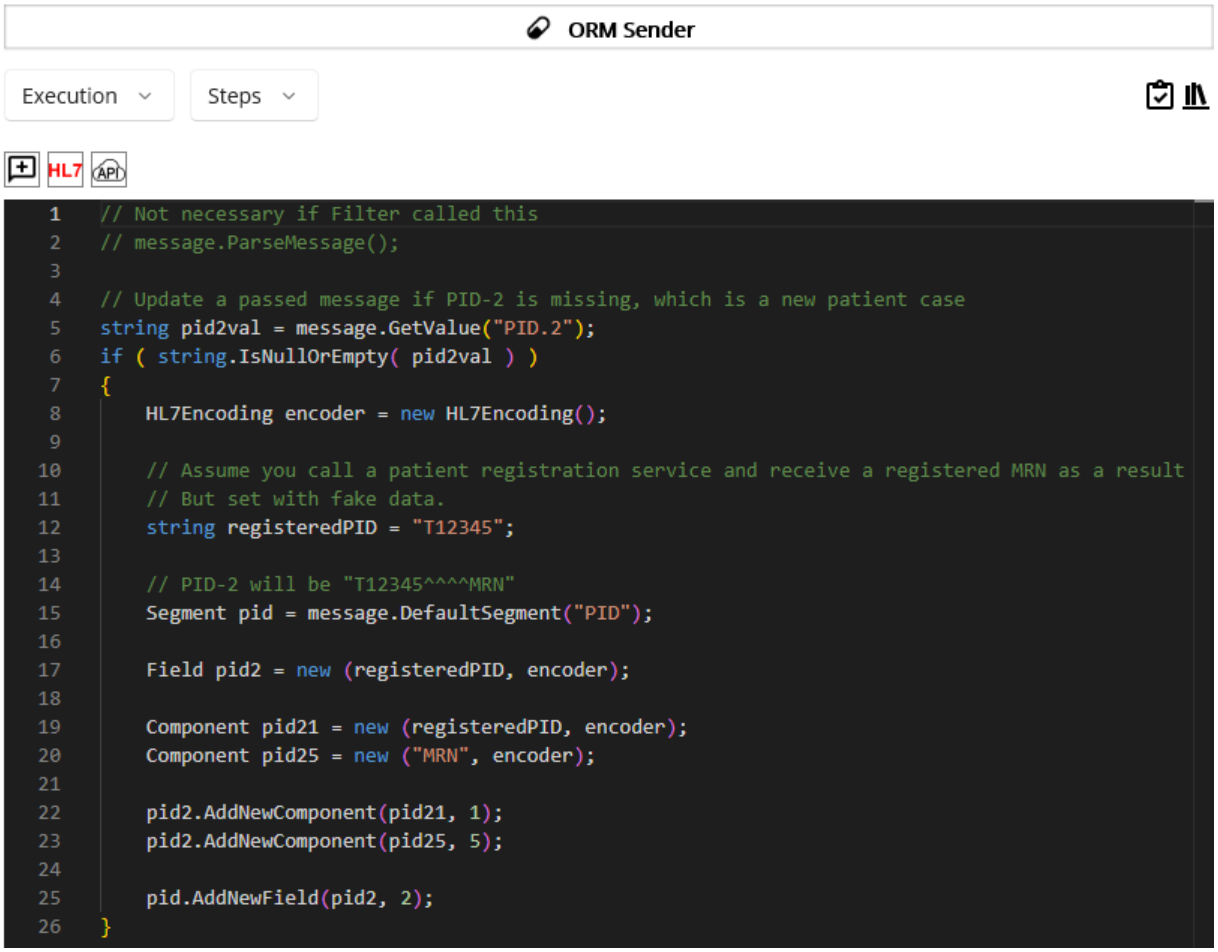


Figure 155.Populate missing PID-2 with a new MRN

As you see in Figure 155, if PID-2 is missing, a fake MRN will be populated in that field, assuming that is a result of an external service call. Other than that, no special code was written. You can easily follow the sample code on HL7-V2 site. Alternatively, you can use the **HL7 Message Generator** popup to validate the code and insert the code you need. As a side note, since the Insert ( ) button will insert the entire code in the popup editor to the main editor, you may have to modify the code a lot in the main editor. Once the verification is successful, you can modify the code by keeping only the necessary code in the editor. And then click the Insert button. Or you can highlight the required code and copy it (Ctrl + C). Then close the popup. Place the cursor where you want to add and paste it (Ctrl + V). See Figure 156 for what you may insert or copy.

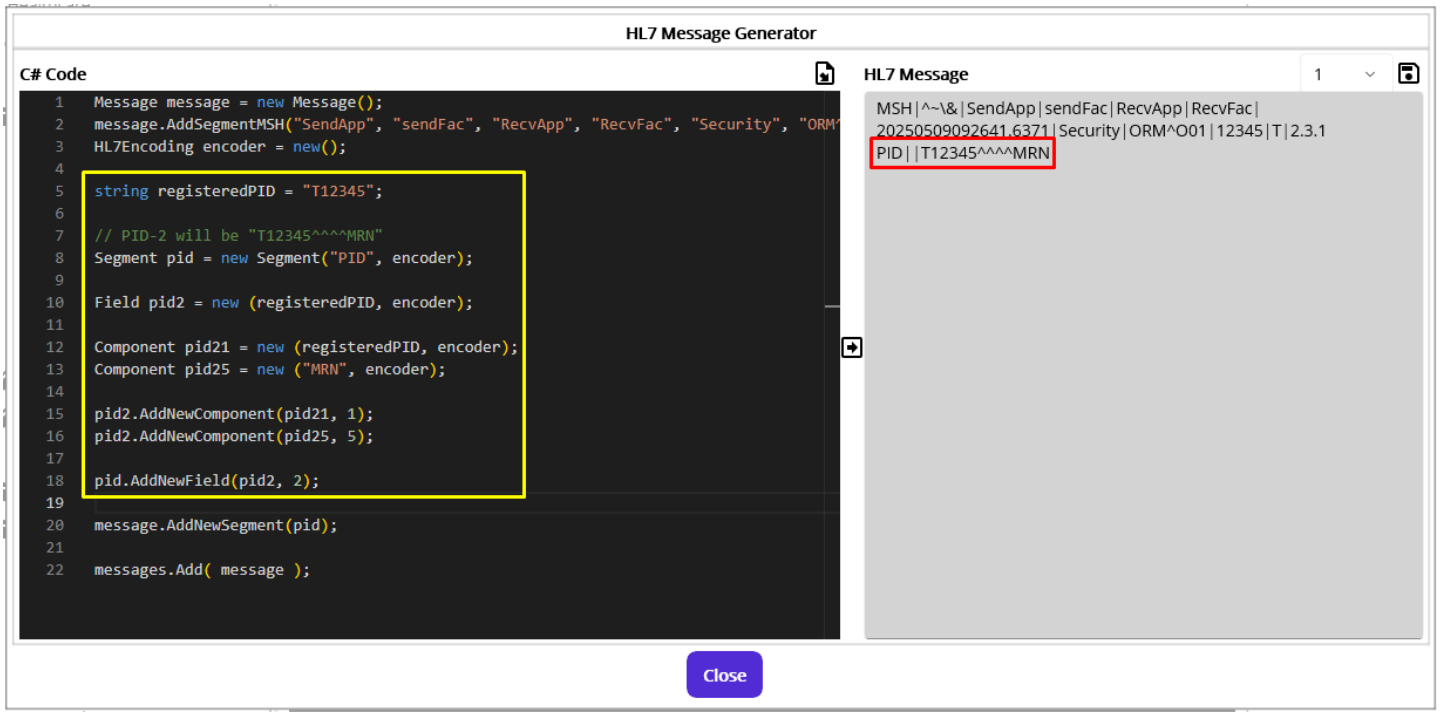


Figure 156. Generated HL7 message

In order to test, update DICOM Reader's Steps code. In this case, just comment PID-2 populating line, as shown in Figure 157.

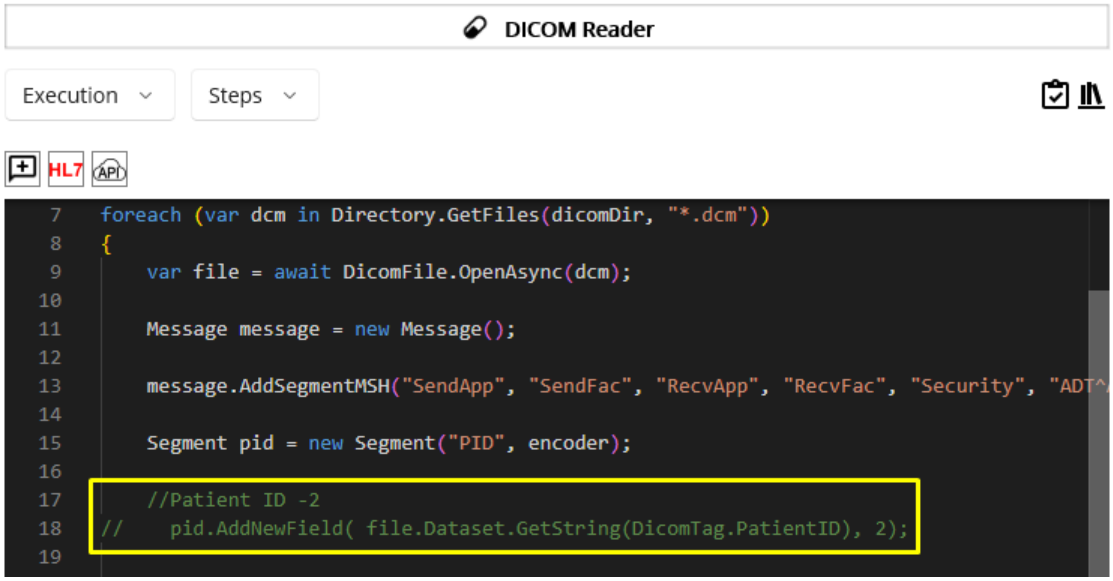


Figure 157.Comment PID-2 line in the DICOM Reader

Finalizer

Once the Filter and/or Steps code finished as Succeed, Filtered, or Error, you can put any follow-up code in the Finalizer view. Although you can put such code in the Filter or Steps code editor, this Finalizer view can give you better insight by separating the code from them. Since the code defined here will be executed anyway after the main logic is finished, you can put the code that executes according to the previous steps' result, i.e., Succeed, Filtered, or Error. In addition, the Finalizer exposes the ACK message received from the client application. So, you can put corresponding code that can be executed according to the ACK's result.

Note: Even if the previous steps were finished with Filtered or Succeed, the Finalizer code may produce an error and overwrite the final result with Error as a result. For example, you generated and sent a valid ORM, and the receiving application sent an ACK

message after processing the ORM successfully. But if the Finalizer code faces an error due to the unhandled exception, the transaction will be saved as ERROR in the database. We will cover this situation later with an example.

Using the service navigator, go to Execution > Finalizer.

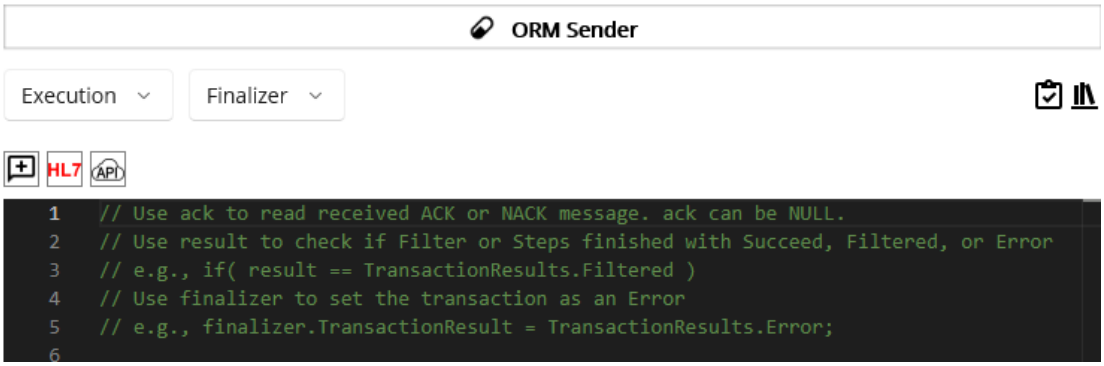


Figure 158.Initial Finalizer code editor

As you see in Figure 158, there are commented lines inside the code editor once you open the Finalizer view for the first time. As mentioned several times, once we send a message, the receiving application will send an ACK message as a sign of the message acceptance. Since the message contains short details of the result, such as AA and text message, we can put corresponding code in the Finalizer editor to process code accordingly. Below Table 21 shows the available variables, *ack, result* and *finalizer* that shown in Figure 158.

Variables	Type	Example	Details
ack	Message	MSH|... |ACK^**R01**|... MSA|**AA**|12345|Message accepted ERR||	An ACK Message object returned from the receiver. **Null** if result is not Succeed.
result	TransactionResults **enum**	if (result == TransactionResults.Succeed)	Result of the Filter and Steps
finalizer	RonResult	finalizer.TransactionResult = TransactionResults.Error; finalizer.ResultMessage = ack.GetValue("MSA.3");	If this is assigned with Error, the main result will be updated as well.

Table 21.Variables in the Finalizer

Once you decided to check the ACK, you may need to look at each segment of the message. Visit https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/message/ACK.html to see the segments of ACK. But in most of the time, you will check the MSA segment (https://hapifhir.github.io/hapi-hl7v2/v231/apidocs/ca/uhn/hl7v2/model/v231/segment/MSA.html), as shown in Figure 159.

Represents an HL7 MSA message segment (MSA - message acknowledgment segment). This segment has the following fields:

- MSA-1: Acknowledgement Code (ID)
- MSA-2: Message Control ID (ST)
- MSA-3: Text Message (ST) **optional**
- MSA-4: Expected Sequence Number (NM) **optional**
- MSA-5: Delayed Acknowledgment Type (ID) **optional**
- MSA-6: Error Condition (CE) **optional**

Figure 159.Fields in the MSA segment(v2.3.1)

Among fields, you will check the MSA-1 field, which is Acknowledgement Code, in most cases. You can visit CodeSystem: acknowledgmentCodes(https://terminology.hl7.org/CodeSystem-v2-0008.html) to see the details. Table 22 shows the value sets excerpted from the site. Among them, AA, AE, and AR will be used mostly.

Code	Display
AA	Original mode: Application Accept – Enhanced mode: Application acknowledgement: Accept
AE	Original mode: Application Error – Enhanced mode: Application acknowledgement: Error

AR	Original mode: Application Reject – Enhanced mode: Application acknowledgement: Reject
CA	Enhanced mode: Application acknowledgement: Commit Accept
CE	Enhanced mode: Application acknowledgement: Commit Error
CR	Enhanced mode: Application acknowledgement: Commit Reject

Table 22.MSA-1 Acknowledgement Codes

Since Maca uses HL7-V2 API, you can utilize HL7 Message Generator popup to see the sample ACK messages based on the generated sample ORM message. You can reuse the code in Figure 130 for the ORM, and use the message object to create ACK messages by calling GetACK and GetNACK methods, as shown in Figure 160.

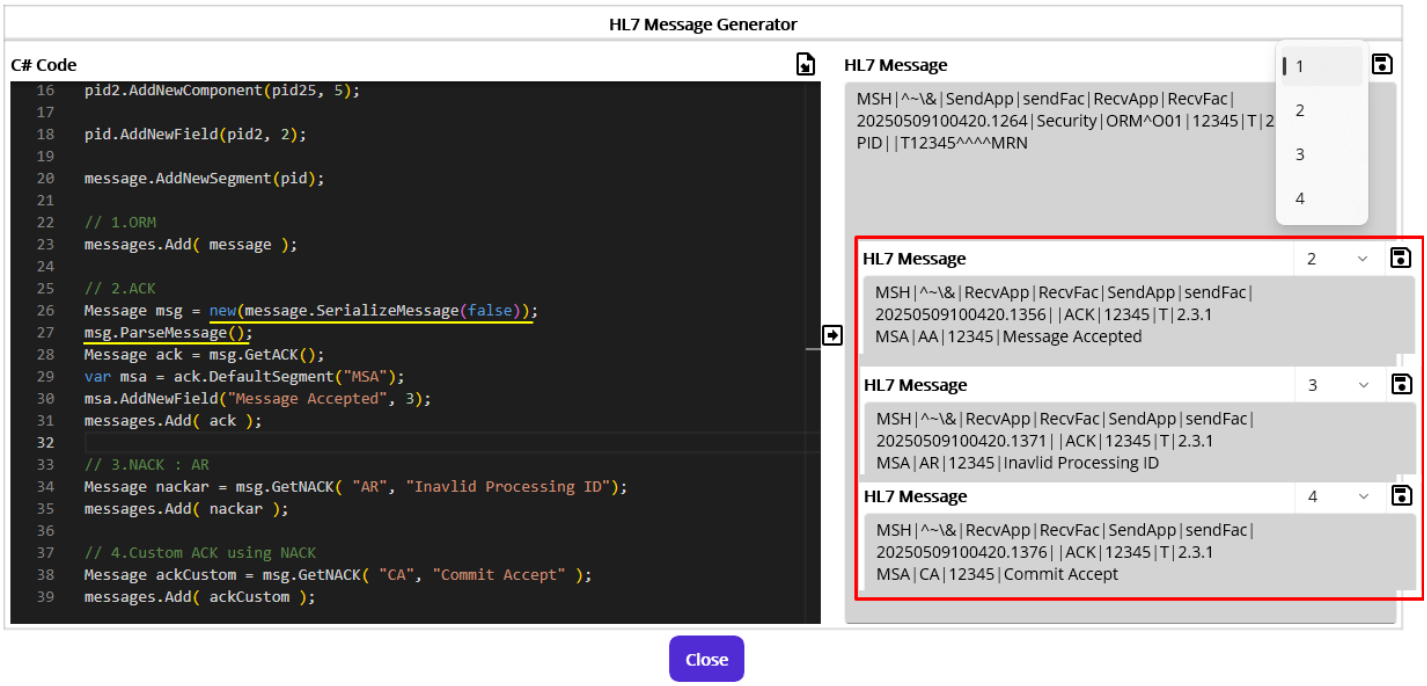


Figure 160.Generated sample ACK messages

As you see in the C# Code editor on the left, there are 4 sections. Each section adds generated Message object to Maca's ***messages*** list variable. By design, once the messages in the *messages* list are validated, all generated messages will be available in the HL7 Message display pane. On the top right corner, there is a message picker next to the save button, as shown in Figure 160. Since we added 4 messages, there are 4 elements that we can pick. Other than the 1st element, which is an ORM, all other messages are ACK messages generated by using GetACK and GetNACK methods. Based on the Table 22, we populated AA, AR, and CA to the MSA-1 field. One thing to note at this point is that MSH-9 is populated with "ACK" only. Sometimes it is required by the client system to populate the field with an event type, such as "ACK^R01," as shown in Table 21. Since the built-in methods don't populate that automatically, you need to populate the 2nd component manually with what your client expects. You can find event type in CodeSystem: eventType (https://terminology.hl7.org/CodeSystem-v2-0003.html).

As a side note, 4th ACK message is not relevant to NACK type because the code is CA, which is the accepted condition. We took advantage of GetNACK method to build an ACK in a shorter way than the 2nd ACK message because the most of the values are almost the same.

Note: Finalizer view doesn't necessarily require an HL7 message generation. Figure 160 is to give you an example of ACK messages that you will receive once you send the HL7 message. ACK generation could be more relevant in the HL7Listener service.

As we saw what we may receive after sending an ORM message, add some code in the Finalizer code editor. One case that we can handle is the returned ACK that can contain error information. See below Figure 161.

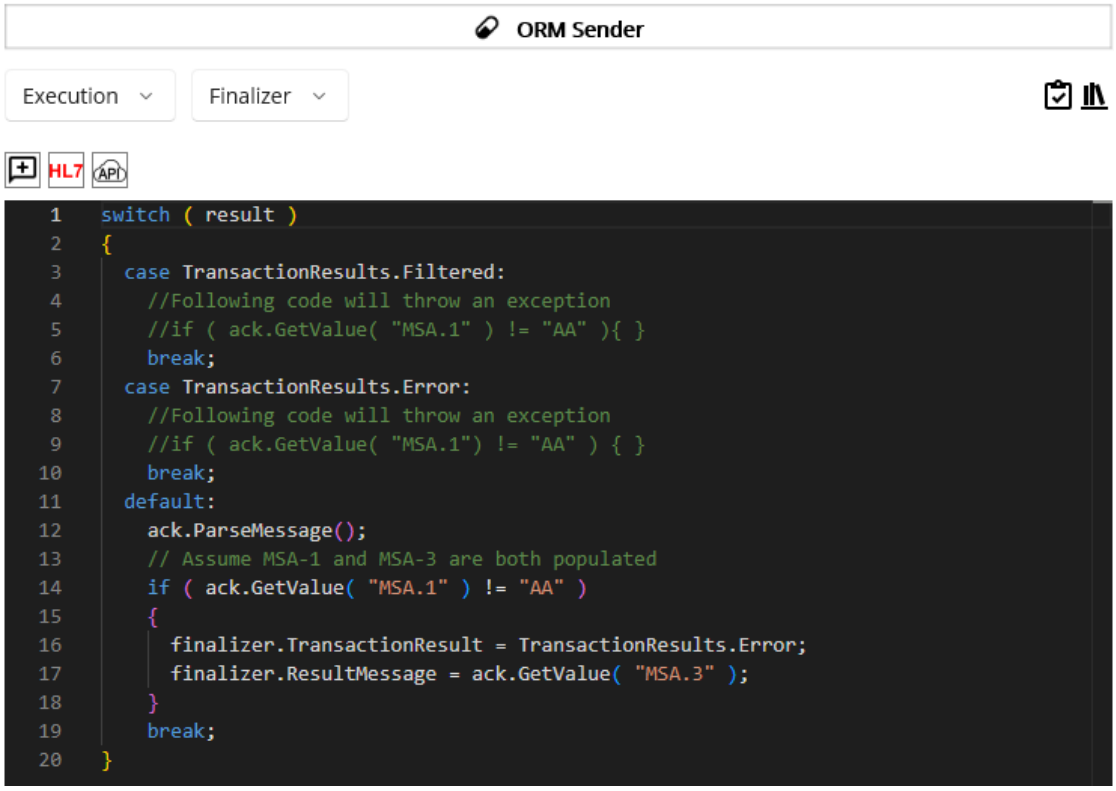


Figure 161.Handles ACK message

What you can see in Figure 161 is the switch statement to handle the result, i.e., Succeed, Filtered, and Error. As you see, the first case is Filtered. One thing to note here is that if you write code that handles Filtered case and it throws an exception, the main result will be overwritten with Error. For example, commented code says that accessing the *ack* will throw an exception because the *ack* is NULL, as you see the detail in Table 21. So, if we uncomment the line and try to access the null variable, an exception will be thrown. However, there is no try-catch block to handle that exception in the sample code. In that case, Maca catches the Exception, sets the main result as Error, and saves the exception to the database. As a result, the Filtered message will be saved as Error, as shown in Figure 162.

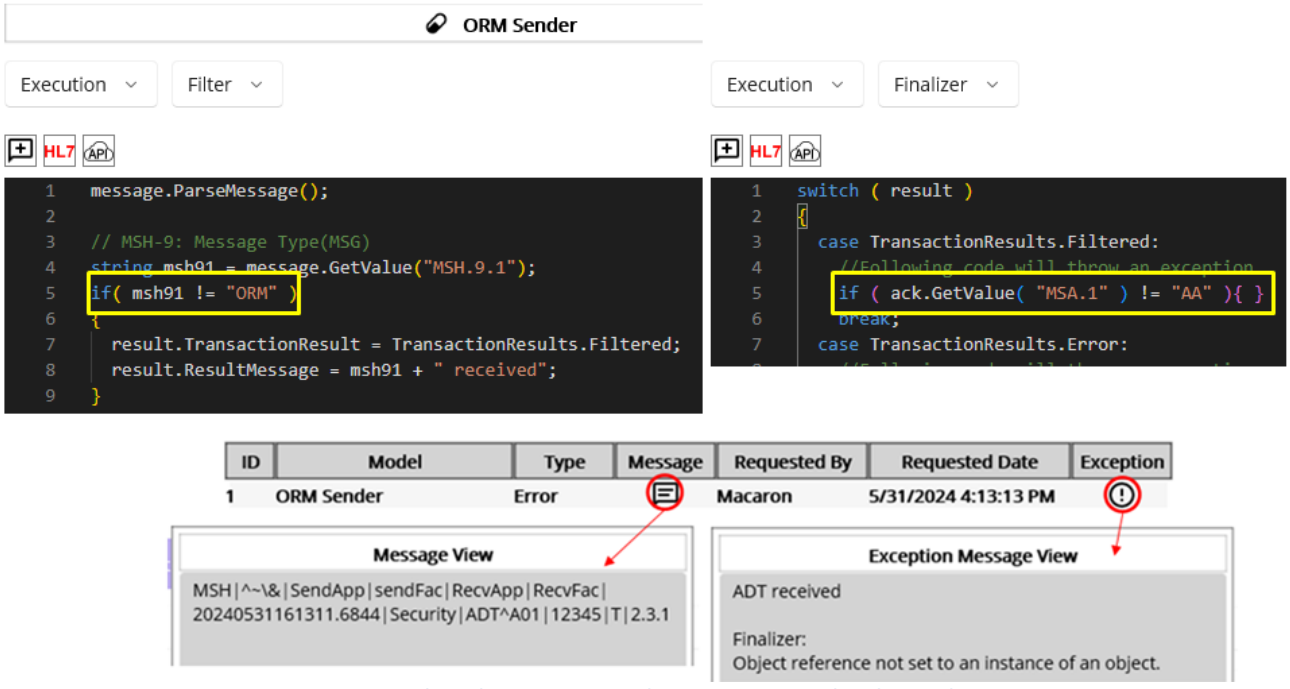


Figure 162.Filtered overwritten with an Error occurred in the Finalizer

Although Maca handles that exception, it is recommended to handle exceptions with a try-catch block if you want to handle more.

Secondly, if you want to handle the Error case, you can add corresponding code in the second case block. But at this time, if you encounter any code error, Maca will save the exception along with the previous exception to the database, as shown in Figure 163.

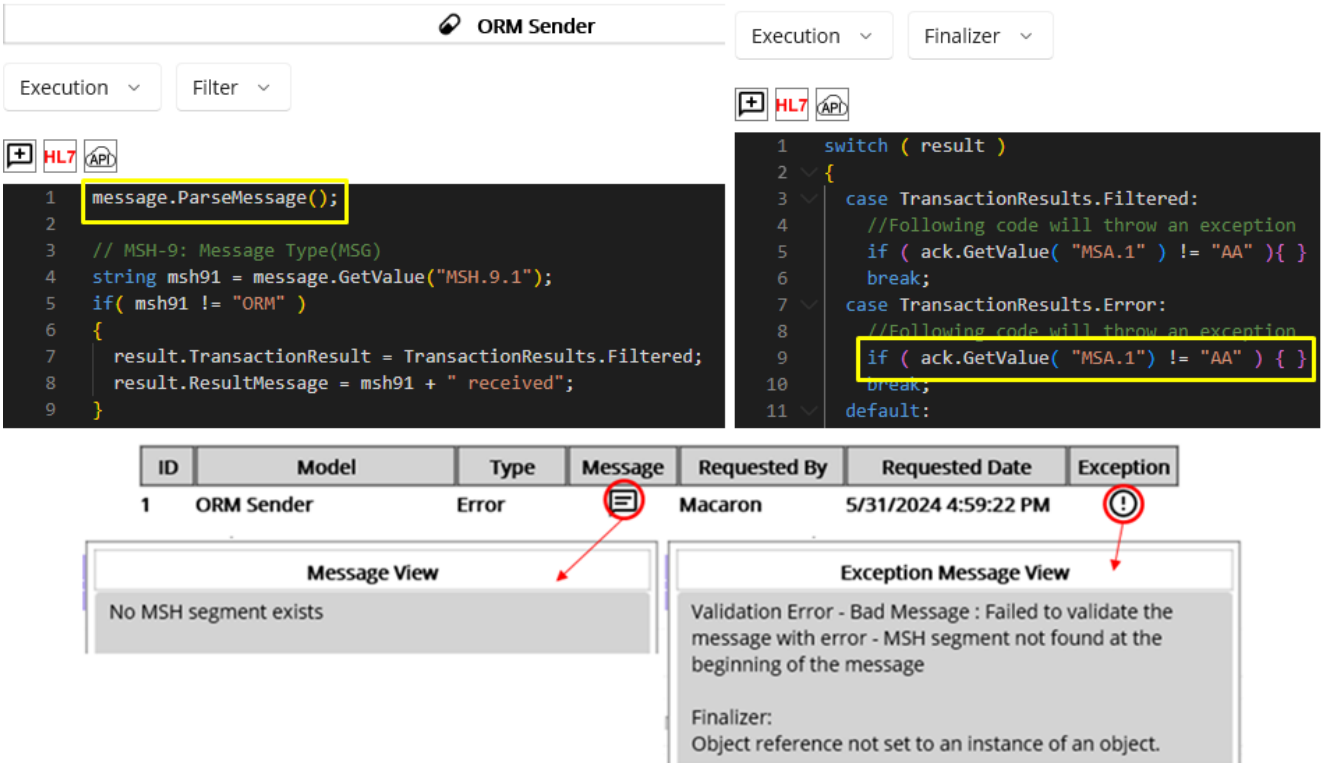


Figure 163.Additional Error occurred in the Finalizer

As we did in Figures 143 and 145, we can create a Message without an MSH segment and send it to the ORM Sender service to make this situation. Since the Message was not parsed and thus marked as an Error, we expect that the error message related to that case will be saved in the database. Then we face another error due to the NULL reference access in the Finalizer. In this case, Maca will append the exception message and save that to the database. So, you can see the additive message, as shown in Figure 163.

Note: By design, the main result marked as Filtered or Error can't be converted to Succeed in the Finalizer. There is no restriction to set finalizer.TransactionResult to Succeed, but that won't affect the main result, i.e., still the result will be Filtered or Error.

Lastly, we added some follow-up code based on the data in the received ACK. As we saw in Table 21, the receiving application can populate the MSA.1 with AA or AR after processing the message. Then we can handle each case by using the related fields. But in most cases, we will handle non-AA cases, as shown in Figure 161. As you already noticed, although the message passed the Filter and Steps conditions, it could be unacceptable in the receiving application. So, it is required to check those cases and set the transaction as Error to save the transaction in the Finalizer. Then we can debug or troubleshoot the issues with that information. To test this case, it would be better to finish HL7Listener part first because we need a returned ACK. Then we can verify the result in the Management workstation once the transaction finished. Keep the code as is, and move on to the HL7Listener service.

HL7Listener

So far, we updated the DICOM Reader and ORM Sender services to test some cases including missing PID-2 population case. We are now ready to send the message to the destination, which is the client application that receives the ORM messages. However, your client may not provide the test application that receives the message right at this moment. So, we need an application that can simulate the client application. For that purpose, you may use a standalone application, e.g., Mirth, which can communicate with the ORM Sender.

As an alternative, Macaron provides the same service, the HL7Listener that was designed to communicate with the HL7Client, in this case the ORM Sender. Although you can create an HL7Listener service on another Macaron server to simulate the client, it is still OK to create the service on the same Macaron server. Since the services communicate based on the IP address and port under the same network, the location will not be a problem as long as they look at the same IP address and port that are not occupied by the other application. So, we can assign the local machine's IP address and port to the HL7Listener, as we assigned them in the ORM Sender. Because the service is supposed to receive ORM messages, we can set its name to ORM Listener.

Testing purpose services

As we decided to create the ORM Listener on the same Macaron server to simulate the client application, we need to pay little more attention from the source management perspective. Once your client's application is ready to test, your test services will no longer be necessary and you may delete from the project to keep only required services. However, we may need them later to test updated ORM Sender that implements new requirements. Then you may import the old one that was stored somewhere and remove it again once the test is done, which could be cumbersome. Or you may decide to keep the test services inside the project and do not load them on the server. However, there is no special way to mark them as testers other than by naming them, like "ORM Listener Tester." And it will be quite tricky to maintain them because the required service's name may contain "Tester" based on its intended purpose. In addition, if the tester services are accidentally loaded on the Macaron server, they will be deployed and started automatically when the server starts or restarts. And they may interrupt other services or generate unexpected errors. Whether you keep the test services in the project or not, you still have to pay extra attention due to this possible reason. And Maca provides a simple but special folder to handle this situation.

Default Testers folder

As you saw, every project has a default folder called "Testers." This special folder is designed to contain all testing purpose services. So, you don't have to assign any words like "Test" or "Tester" just to segregate the service from others all the time. And you can freely assign "Tester" to any of the required services located outside of the Testers folder. In addition, any services inside this folder will not be automatically deployed, even if they are accidentally loaded on the Macaron server. As a result, you don't have to store the testing purpose services somewhere else, and thus you or your successor can easily catch up on the service update later.

Create an HL7Listener

Since the ORM Listener is for testing purpose, select Testers node from the project explorer. Then click the Create Project Model button as we did in the HL7Client section, as shown in Figure 164.

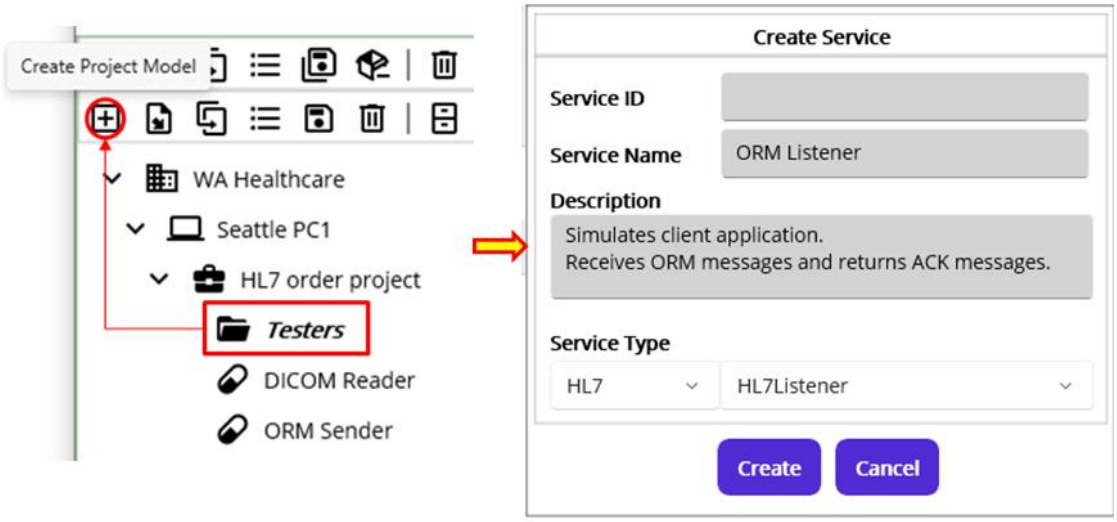


Figure 164.Creates testing purpose H7Listener service under the Testers group

At this time, the Create Service popup showed up instead of the Create Group or Service popup. See Figure 136 for comparison. Since the group model can hold Macaron Service model only by design, the popup doesn't have a Group radio button option. In the Service Type section, select HL7 and HL7Listener from the pickers. Type necessary information and click the Create button.

Note: To differentiate the tester service, its icon ( ) is different from the regular service's one ( ).

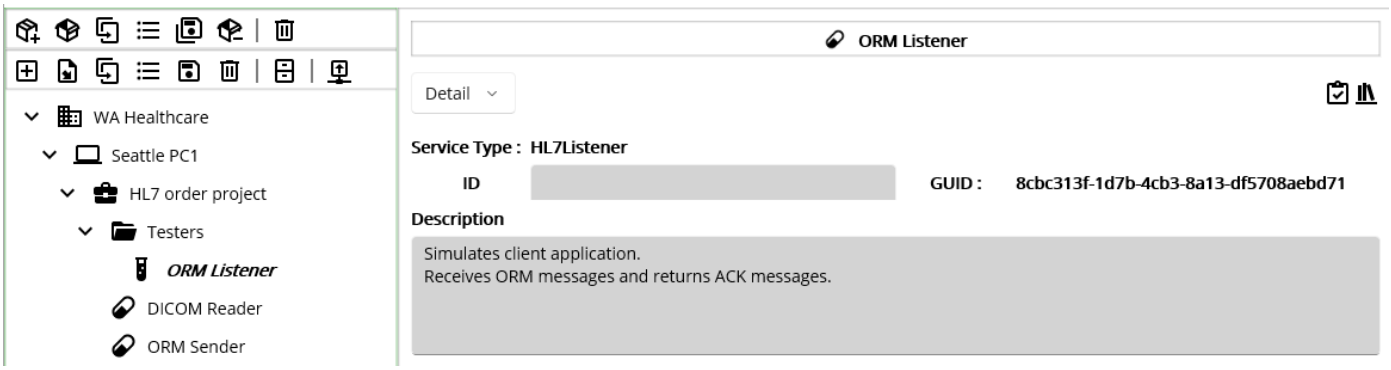


Figure 165.Created ORM Listener tester service

Configure the listener

Once the ORM Listener service was created, go to Settings > Config view using service navigator. As we set localhost and 8081 for the target destination in the ORM Sender service, the ORM Listener, which is a destination, needs to have the same information. Please type **localhost** for the IP Address and **8081** for the Port in the HL7 Listener Setup section, as shown in Figure 166.

Note: Port 8081 could be occupied by the other application. Please use the other available port, i.e., not occupied, in that case.

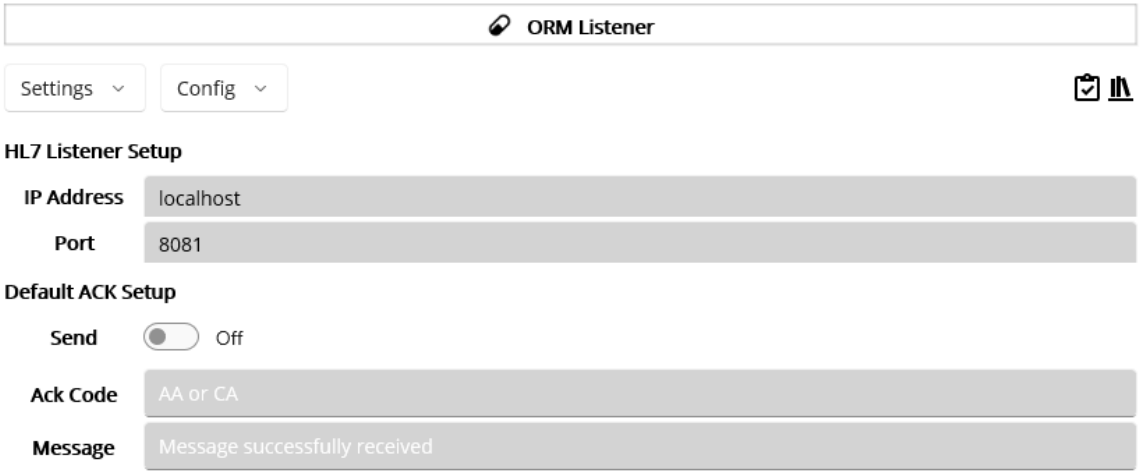


Figure 166.ORM Listener configuration

Default ACK message

In general, the HL7 message sender will expect the ACK message once they send the message. In our case, the ORM Listener needs to send an ACK message as a response once the ORM message is transmitted from the ORM Sender. As we saw in the ORM Sender's Finalizer, the MSA.1 field could be populated with AA, AR, or CA in the ORM Listener. In a real-world case, your client may provide sample ACK messages that you can test with before the test environment is ready. Or they may provide what kind of possible values you will see in the ACK messages. So, it is required to consult with them about what kind of the data needs to be populated in the ACK message for both ends. In certain cases, your client may want a pre-defined data, no matter what the result is. In other words, they just want to check whether the message was delivered, for example. If that is the case, you can set that value in the Default ACK Setup section, as shown in Figure 165.

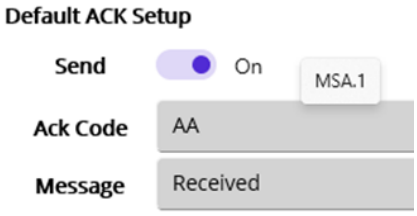


Figure 167.Default ACK Setup

In most cases, the default ACK message code will always be AA or CA because it will be a simple sign of a delivered message. However, it is still required to ask them about the data. Once you configure this value, Maca will generate the ACK and return it to the original message sender. Please check #2 in Figure 160 for the generated sample ACK message with AA code. For now, **turn the Send switch on**. We will turn this on and off few times later to test some cases. Then type values for Ack Code and Message, as shown in Figure 167. As you may notice by hovering the mouse pointer over the input field, the Ack Code value will be populated in MSA-1, and the Message value will be populated in MSA-3.

Note: Once you configure the service to send a default ACK, any custom ACK you create in the Filter or Steps view will be ignored. So, please make sure to turn this configuration off when you want to send a custom ACK message.

Deploy and Undeploy

Since this service simulates the client's application, we don't know what exactly they do or require when their application starts or stops. What we will focus on in this service is the message exchange, and thus we can skip this part. But as we did in the DICOM reader, you may put any code that's necessary for testing purpose.

Filter

Move to Execution > Filter view using service navigator. As you see in Figure 168, there are commented lines that contain some codes when you open the Filter view in the HL7Listener service for the first time.

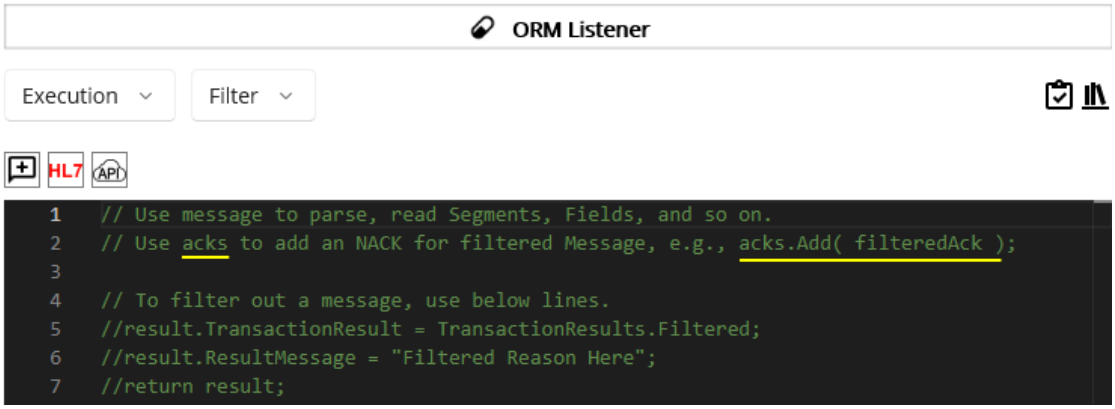


Figure 168.Initial HL7Listener Filter View

As you saw the *messages* variable in the HL7 Message Generator popup, the Filter in the HL7Listener has a similar variable ***acks***.

So far, we saw one HL7 message in a transaction, which means that the application is transmitting one message in the MLLP frame. In other words, the ORM Sender sends one HL7 message to the client's application every time. In some cases, the sender can send multiple messages in a transmission, and the receiver may need to send the same number of ACK messages as a response. We will go through the structure and usage of the multiple messages in detail later with examples. But for now, let's assume that we send one message per transmission to the client's application. Although the ORM Listener receives one message, it is still OK to use the *acks* variable to send one ACK message. See Table 23 for the detail.

Variable	Type	Usage Example	Detail
acks	List<**Message**>	var nack = message.GetNACK("AR", "Rejected"); acks.Add(nack);	Instantiated per transaction. Designed to send multiple ACK messages in

			case of multiple messages in the received data, such as a batch message. The acks will be ignored if Send Default ACK is turned on.

Table 23.The acks variable in the HL7Listener's Filter

Note: Although we used the term NACK, actual message type is ACK. So, creating a NACK means creating an ACK with a negative result value, like AR or CE.

Since we are in the Filter view, what we will do here is filter out invalid messages first. Then if any defined filtering condition is met, we will filter out the message as well. As you may notice, this is exactly the same as what we did in the ORM Sender's Filter. The only difference is a negative ACK that we have to create once we find any invalid or unacceptable message. So, we can reuse the code in Figure 148 and extend it to handle the NACK message. As a side note, please note that we will not use GetACK() that is for successful message. Although we can create a NACK with GetACK() by populating the MSA.1 with a negative value, GetNACK() makes code more relevant and concise. So, if you are comfortable with both methods, please use one of them as you prefer. Following Figure 169 shows a code implementation of message filtering and corresponding NACK message creation as a result.

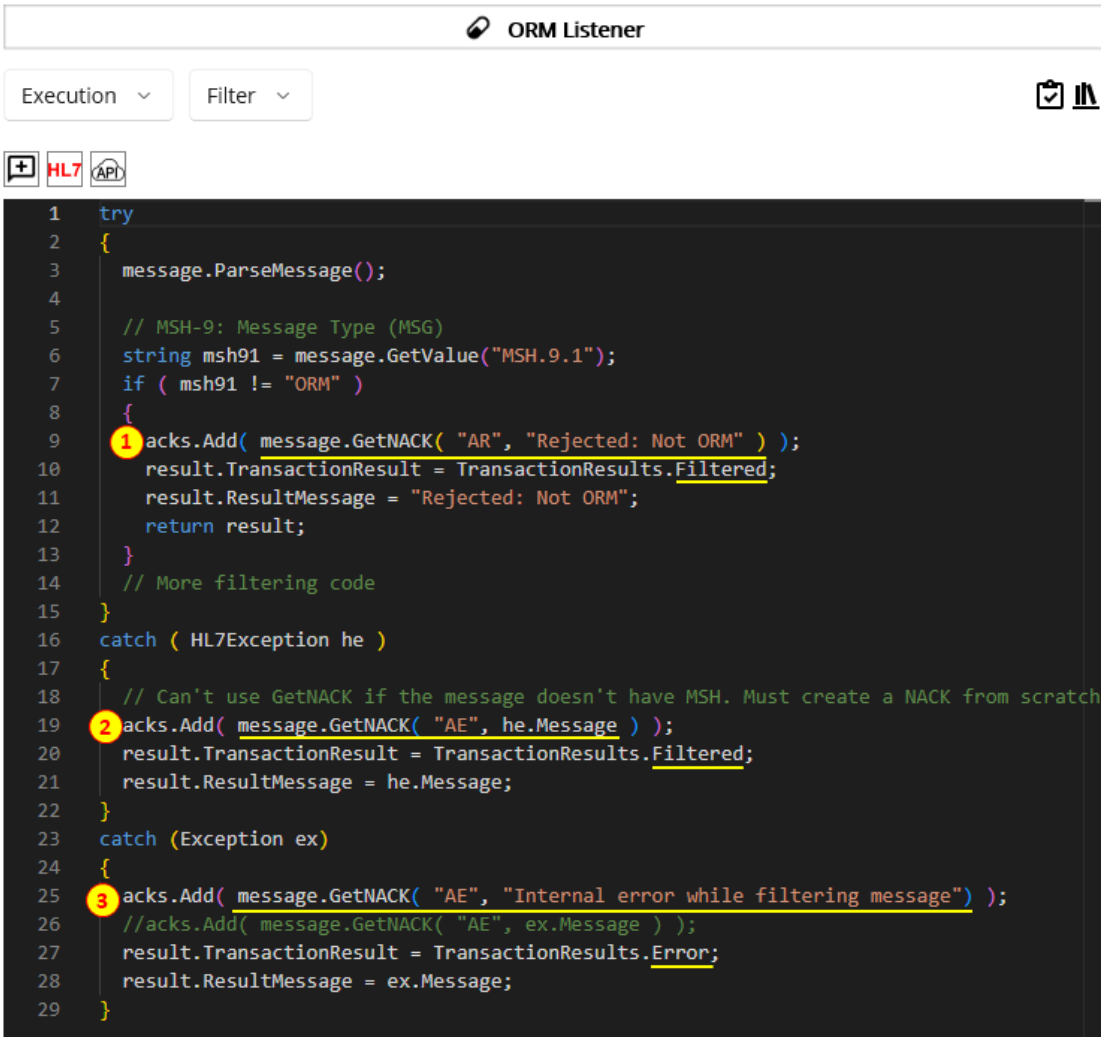


Figure 169.HL7 Message Filtering example

Like an example in Table 23, creating a NACK is quite straightforward.

1) Once the ParseMessage finished without error, we can use GetNACK method to create a negative ACK if any defined condition is met. Since we filter out non-ORM message, we can set the message as rejected, i.e., AR, as shown in Figure 169.

2) If the ParseMessage fails, i.e., throws an HL7Exception, we can use GetNACK again and populate corresponding values in the HL7Exception catch block. In this case, we may set the message as Error, i.e., AE.

3) Lastly, if we face any other exception, like code error, we can do the same thing in the Exception catching block. You can set the reason value as either a predefined value, like "Internal error while filtering message," or ex.Message that best fits. Although the code sets AE for the exception case, you can set AR or CR if that is more appropriate for the situation.

Note: One thing to note is the received message may not have a MSH segment, which is an edge case. In that case, ParseMessage fails and throws an HL7Exception. You can revisit Figure 143 and 145 for this issue. However, #2 in Figure 169 will fail as well because GetNACK reads MSH segment, which is null, and throws an exception. Although Maca will catch this case outside the Filter, the result will not be as you expected. This is an extremely rare care, but if you think that can still be possible, create an ACK from scratch, i.e., use the Message class, like #2 in Figure 160.

Once the NACK is created, we need to add the message to the ***acks*** list variable, as we did with the *messages* list variable.

Then we need to exit the Filter context. If there is more code below the line and you need to exit, use "return result;" that doesn't proceed with the rest of the code. Otherwise, just setting TransactionResult and ResultMessage will be enough, like #2 and #3 in Figure 169.

Exception Handing

In each view, such as Filter, any unhandled exception will be caught by Maca. As you saw in Figure 86 and 145, if there is no try-catch block in the code editor, any exception thrown by the written code at execution time will be caught as Exception by Maca and the transaction will be marked as an Error. If you want to handle any specific exception, you can use the try-catch[33] statement with exception filters and put the corresponding exception handling code in each catch block, as we did for ParseMessage() in Figure 169. (For exception handling, visit https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/exception-handling) As a side note, the exception mentioned here is an error that occurs at execution time, i.e., when the service is running. On the other hand, the code verification (📋) handles the error at compile time, i.e., verifies the code was written correctly. So, whenever you modify the service, please make sure to verify the service at all times.

Note: Although the example uses exception filters, i.e., HL7Exception and Exception, you may use Exception only if no specific exception needs to be caught.

Steps of code to handle received messages

Since our main goal is to receive an ACK after sending an ORM, we may not need any specific message handling logic in the Steps view. In addition, we didn't receive any specific requirements from the client yet, so adding some code to the Steps would be just code practice and irrelevant to the client's requirement. However, using the Filter wouldn't be enough to create more test cases because most of the issues will happen while handling message contents. As we don't have any specific requirements to implement, we will not dig into the message deeply. Instead, we will check a few fields that you can use as a source for extra test cases as well as service development. That will give you an idea of how to interact with the sender from the receiver's perspective because you will work with the HL7Listener in the future as well. Using service navigator, go to Execution > Steps view.

In the Filter, we added conditions to create a NACK message only. In the Steps, we can create either ACK or NACK based on the condition we define. But at this time, we don't have to validate message because we called ParseMessage(), which validates the received message in the Filter, and the Steps will be skipped if the message was filtered. What we need to do here is read the content, e.g., field, of the message and do some work with it. If all required values are passed and correct, then we can create an ACK. Otherwise, we need to create a NACK. As you already noticed, the structure that handles exception is similar to the Filter's one. So, we can reuse the code and add content reading code, as shown in Figure 170.

[33] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements

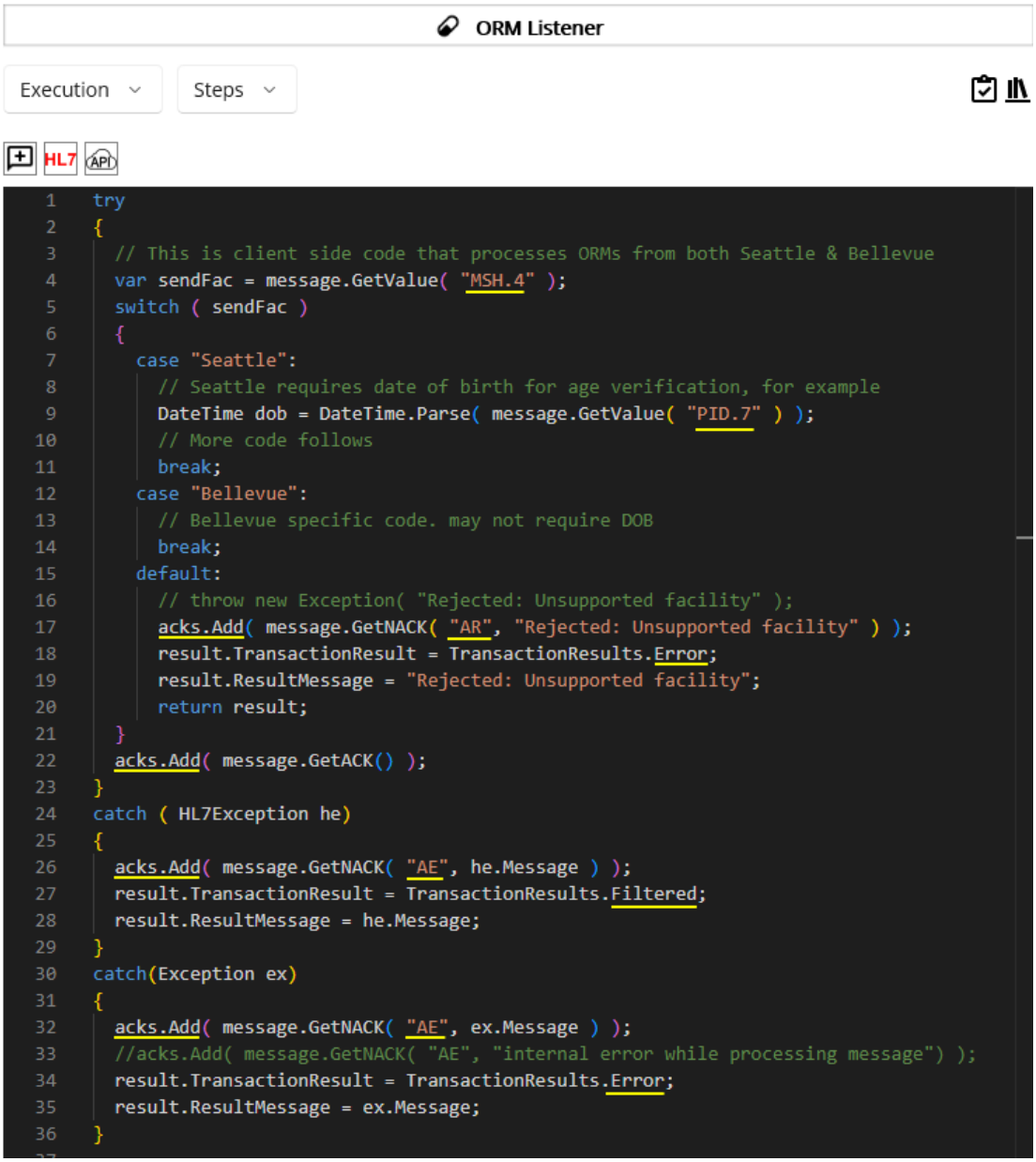


Figure 170.ORM Listener's internal code

As mentioned, we are not going to check parts or entire content. Instead, we will focus on the two fields, MSH.4 and PID.7. Yet this will give you helpful examples of how to handle values and exceptions in the received HL7 message in the Steps view. Based on these fields, we will create hypothetical test scenarios and corresponding test cases. So, the ORM Sender will send messages related to this code and receive the results, i.e., ACK messages, to conduct tests. See below key points.

Note: The ORM Listener is simulating the client's application. Internal code in the Steps will not be what we have to consider. For example, we are not interfacing with the Bellevue location. But WA Healthcare could check each location in their system. The code here is for test-case scenarios as well as code handling purposes. The real-world scenario and practices will be totally different.

First, the code checks MSH.4 field, which is a sending facility. Since we are working on Seattle's project, MSH.4 could be our client's internal value, such as Seattle, as shown in Figure 170. Let's assume that the Seattle location's medical service specifically checks the patient's date of birth to verify age-related symptoms, such as type 1 and type 2 diabetes. In order to parse age, DateTime's Parse method was used to convert the string value in PID.7 to DateTime. As a side note, we will check the exception case related to the Parse method. So, just one line of code will be enough for testing but you can add more code for your own test cases.

Second, if the MSH.4 is neither Seattle nor Bellevue, the code creates a NACK message with “AR.” Although no new location, such as Redmond, is planned to open soon, the client may need to handle inappropriate value for code maintenance purpose. This will be tested in the examples later.

Lastly, if there was no issue found, the code creates the ACK message, and the Steps code ends.

While the code executes, if there is any exception related to a HL7 message reading error or code error, each catch block’s code creates a corresponding NACK message and sets the transaction accordingly, as shown in Figure 170. Although each catch block creates the same AR value, you can redefine AR for the HL7Exception and AE for the general Exception. Or you can keep one Exception block only if the NACK is always AR or AE. So, choose whatever option fits better for your situation.

Keep this code and move on to the next section.

Finalizer

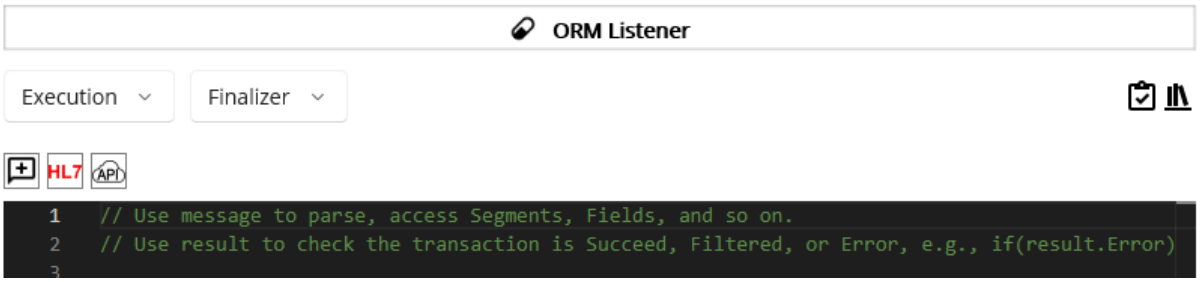


Figure 171.Finalizer in the ORM Listener

Since this is called whatever the transaction result is, you can add extra code for further testing. But keep this code empty because what we needed for the ORM Sender testing is pretty much implemented in the Filter and Steps.

Testing Services

Finally, we finished the implementation of the first requirement. What we need to do next is test the ORM Sender service based on the code that we added, as you expected. But at this time, we will not modify DICOM Reader or ORM Sender to conduct a test. Although we tested some cases while implementing each service and the results were saved in the database, it is hard to track how the test was conducted. From an error reproduction perspective, it will be quite difficult to repro the exact same error because we may have to modify finished service and redeploy several times. In addition, if we have to do the same test again once a new requirement is implemented in the service, i.e., a regression test, it would be really painful to modify the specific parts of the required service one by one manually and revert. It will be inevitable to modify services while debugging or fixing errors, but that will not be a good approach in terms of conducting tests. Instead, we will take a different approach that uses a dedicated service to perform that role.

Note: As you may notice, Maca proto doesn’t provide unit testing internally. So, there is no specific way to thoroughly test the Filter, Steps, and Finalizer separately. In addition, it doesn’t provide any test automation methodology to apply. This tutorial introduces the basic testing methodology by using the CodeRepeater, which feeds the test data to a target service. Still, test results need to be verified manually, but the entire testing procedure demonstrated here will give you an idea of how the service could be tested. Once you finish this tutorial, we believe you can extend the service and define your own test methodology with your own ideas to test each service.

Service Tester

As we have a Testers folder, we can take advantage of that again to test other services. But at this time, we will create a service not as a simulator but as a tester. What this service will do is feed test data to the target service, i.e., the service to be tested. In our case, the DICOM Reader generates ORM messages and feeds them to the ORM Sender. In most cases, you may not find various DICOM files that filled with the data that your test cases need. So, in some way, you may have to create your own DICOM files filled with test case data to conduct tests. With the help of the fo-dicom library that was introduced in the DICOM Reader section, you can

generate multiple DICOM files by populating Tags with the value you need. Then let the DICOM Reader reads all DICOM files that you generated.

But for now, we need to focus on the message exchange with ORM Listener service. So, we will focus on the ORM Sender service that sends the ORM messages and receives ACK messages. As we do not use the DICOM Reader to produce HL7 messages, we need another service that does the same work.

Create a service tester

Select the Testers node and click the Create Project Model button. Keep the Service Type as CodeRepeater and then type the information, as shown in Figure 172.

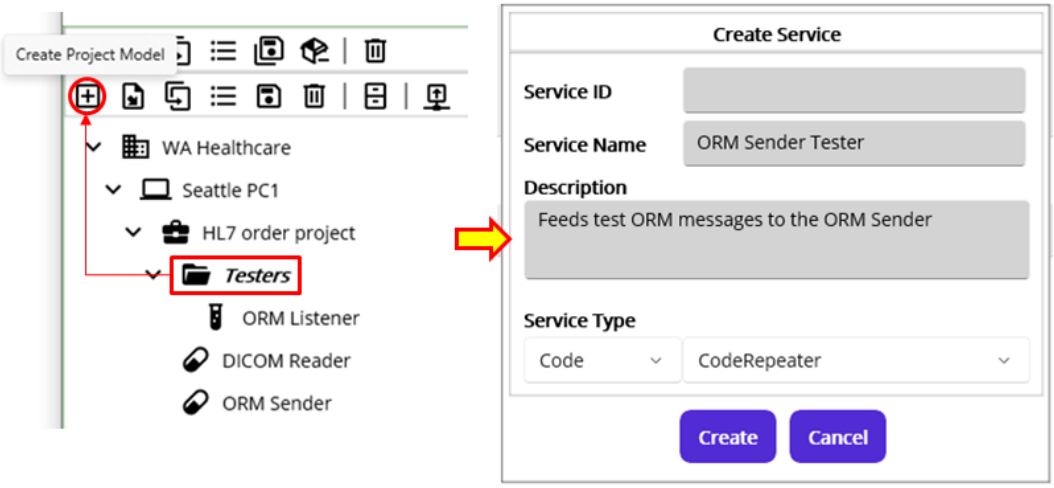


Figure 172.Create a service tester

It is obvious that the service under the Tester folder is for testing purposes, but we can append Tester to make its purpose clear. Or, you can name the service differently, like “Test ORM Feeder.” So, freely set the service tester’s name, whichever fits better.

Note: As mentioned, the Testers folder and its children will not be auto-deployed when the server starts by design and thus will not affect any other required services that exist outside this folder. So, whenever you think a service is not required, always put it in the Testers folder.

Edit Config

Since this service repeats the execution based on the time interval, we need to set the interval little wider than the DICOM Reader. Because we are not repeating the test again and again, we can set the interval to 30 minutes to make sure not repeat soon again. Once the service starts, it will immediately execute the code. Then we can check each service’s results in the Observer > Transaction as we did previously. After that, we can stop or undeploy the ORM Sender Tester.

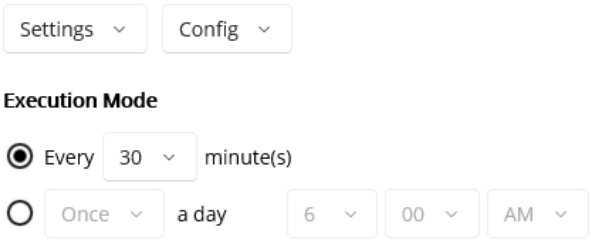


Figure 173.ORM Sender Tester’s Execution Mode

Edit Deploy

For now, we don’t need any specific work before the execution. So, please keep this empty. But you can add some code if necessary.

Edit Undeploy

Like the Deploy, keep this empty.

Edit Filter

As we saw, this is not available.

Edit Steps to generate test data

This is the core part where we need to put test data generating code. Previously, we modified the DICOM Reader and the ORM Sender to handle some issues before sending ORM messages. Then we restored the code to its original state after the tests. In the ORM Listener service, we added some code to process received messages to send ACK messages. So, we need to combine all previously used code here to conduct a test. But before we write code, let's list possible test cases that clarify what needs to be tested based on the test scenarios that we can consider.

Test scenarios

The first scenario is that the client always sends the same ACK message, i.e., AA with a fixed message. As we set AA and Received in the Figure 167 for the Default ACK Setup, we can expect the returned ACK with that value for every ORM message we send. Based on this, the list of test cases will look like follow.

ID	Test Case	Input Data	Expected	Result
1	Non ORM message	ADT message	ORM Sender filters the message	
2	Missing MSH.11 value (Required by HL7)	Empty value in MSH.11	ORM Sender filters the message	
3	Missing PID.7 field for Seattle	No PID.7 field in PID segment	ACK with AA and Received.	
4	Missing PID.7 value for Seattle	Empty value in PID.7	ACK with AA and Received.	
5	Sends a valid ORM message	Valid ORM message	ACK with AA and Received.	

Table 24.Scenario 1: Test cases with Default ACK

As you see in Table 24, we defined 5 test cases based on the first test scenario, which is the default ACK message. This is a really basic test case list that could be extended with more columns, such as test steps and environment, but will give you enough information about the testing.

Among them, one thing to note is the Input Data column. The short description, e.g., "ADT message" or "Valid ORM message," can show you what needs to be sent. But once you make a separate test case document, it will be more helpful if the actual HL7 message is there. So, you can copy and paste the input data to the service tester when testing. To make the table concise, we intentionally put short description. But in the Steps view, entire HL7 messages will be listed.

The other thing to note is that the scenario is more likely the ORM Listener-centric tests. Since the ACK is always the same, what the ORM Sender can check is whether the ACK is sent or not because the ORM Listener doesn't expose any hint of what was happening while processing the received ORM message. However, even if the result is all the same, we have to conduct each test case to see if both sides work as designed and expected.

The last thing to note is that the field can either not exist, i.e., NULL (#3), or empty (#4). Following table 25 shows the difference.

ID	Possible PID.7 cases	PID segment example	In case of PID.7 accessing
1	Field not exists	PID||100|||DOE^JOHN	throws HL7Exception
2	Field exists but empty	PID||100|||DOE^JOHN|||	returns ""

Table 25.Field accessing issues

As a side note, while you create an HL7 message, the field separator "|" will be auto-populated ahead of the last field that you added. Once you created PID segment and added PID.5 only, the expected value will be "PID||||DOE^JOHN" similar to the #1 in Table 25. So the rest of the fields starting from PID.6 won't be available. Please visit HL7-V2 site for more detail to avoid any null field accessing issue.

Note: Above test cases are not real-world test cases. Some are intentionally added to make you familiar with the HL7 message handling, i.e., code practice. Based on the objectives that we defined in the scenario 1 introduction, PID segment related cases are more relevant to actual test cases. Along with that, some test cases, such as no MSH segment, are not included in this test set. Since we are focusing on general cases, you can implement such edge cases later once you feel comfortable with the service development and testing. In addition, populating PID.2 in a new patient case is also excluded because we need a separate service in action to receive the patient ID. So, testing PID.2 populated with a fixed fake patient ID could be meaningless unless you create another test service that generates a new testable patient ID each time.

The second scenario is the opposite situation that the client sends custom ACK messages based on the result of the received ORM messages. As you may notice, above 5 test cases can be reused but the results will be different, as shown in Table 26.

ID	Test Case	Input Data	Expected	Result
1	Non ORM message	ADT message	ORM Sender filters the message	
2	Missing MSH.11 value (Required by HL7)	Empty value in MSH.11	ORM Sender filters the message	
3	Missing PID.7 field for Seattle	No PID.7 field in PID segment	NACK with AE and HL7Exception Message.	
4	Missing PID.7 value for Seattle	Empty value in PID.7	NACK with AE and Exception Message.	
5	Sends a valid ORM message	Valid ORM message	ACK with AA and Processed.	

Table 26.Scenario 2: Test cases with custom ACK

As you saw, there is no change in the test result for #1 and #2 because the messages were not sent for both scenarios. So we can focus on the rest of the cases that highlighted in yellow in this scenario. As we expect, test case #3 and #4 should return NACK and #5 should return ACK with AA. Unlike the first scenario, this is more close to the ORM Sender service test. Based on the above two scenarios, we can implement test message generating code, as shown in Figure 174.

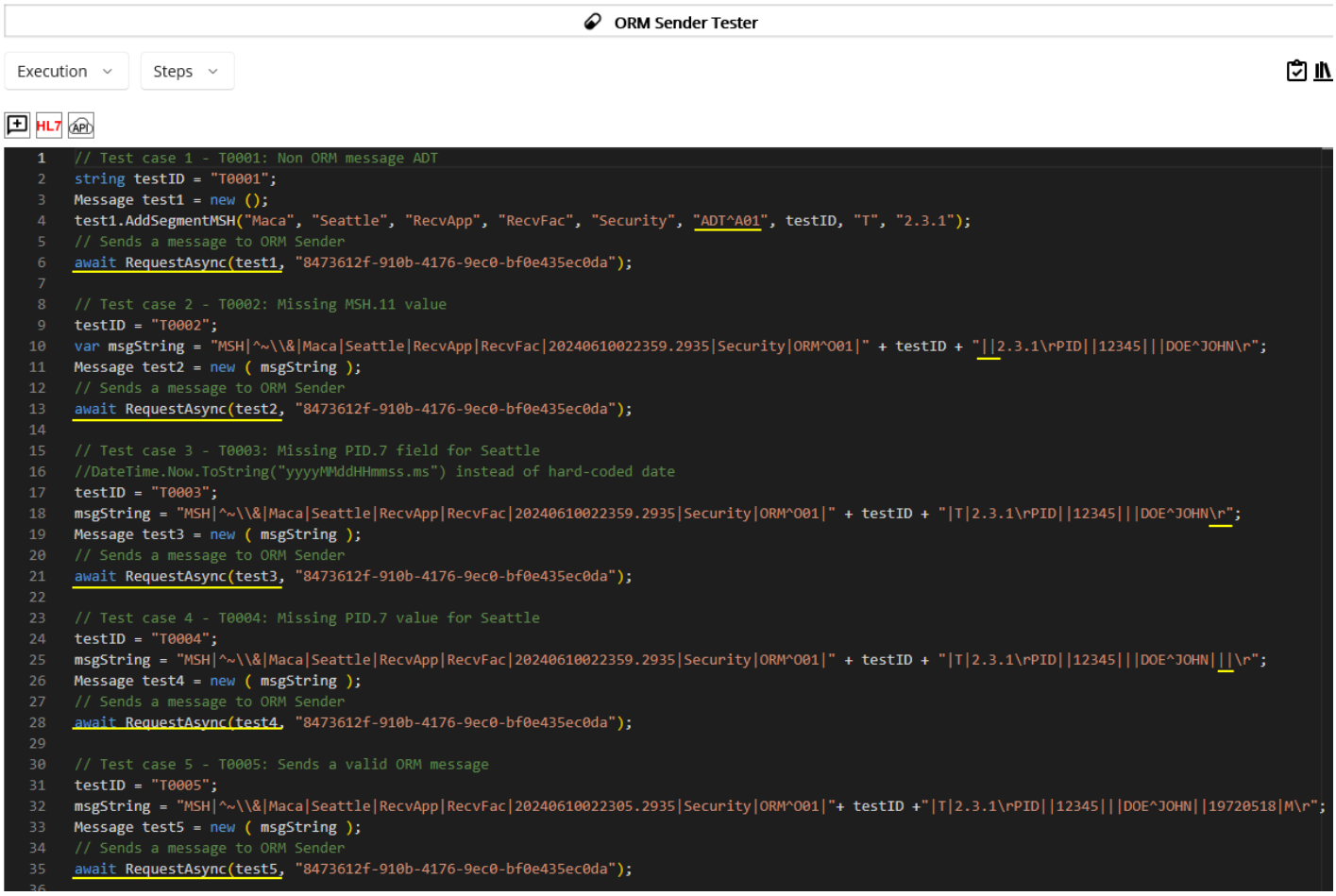


Figure 174.5 test cases for ORM Sender

As you saw 5 test cases in Figure 174, you will notice that there are 3 common codes; **testID**, **Message**, and **RequestAsync**.

First, the variable **testID** will mark the transaction as a test case. Usually, the MSH.10 will be assigned by the system with a trackable ID so that the message can be identified with that value for cross-check purposes. Since Maca saves the MSH segment only, we may use one of the fields to track the test case. As you see, we assigned the first test message's MSH.10 with "T0001" and kept increasing to "T0005," which is the last test case.

Second, each test case creates the **Message** object, which is an ORM, and sets its values with the AddSegmentMSH method or the constructor. If you want to micromanage the values, then use AddSegmentMSH first and add segments along with the fields, as we did in Figure 155. Otherwise, you can use the constructor by passing the entire message with a few different values, like test case from 2 to 5. Choose either way that fits better and is shorter for the testing. But when you decide to use AddSegmentMSH, please populate all arguments because it throws an error if any of the arguments is empty. So, you can't use the AddSegmentMSH for test case 2.

Lastly, once we finish the test message to send, we need to **map the message** to the target service, i.e., the ORM Sender service. Click the Add Service Request Code ( ) button to select the target service. Select the ORM Sender and click Add button. Once the code is added, modify the value with the message you created, e.g., test1. Then copy the line and paste it into the next test case, then modify it with test2. You can keep repeating till test 5. When the service tester starts, it will enqueue the test messages to the ORM Sender. One thing to note is that you need to call this method five times, i.e., for every test case.

Along with that, there are underlined parts in red in the test messages that indicate what needs to be tested. You can create your own test cases by populating or removing specific elements like above example. We just tackled few fields, but you can freely add more test cases related to the segments or fields later.

As a side note, we are not validating the message here because the receiving end needs to do that. If you use HL7 Message Generator popup to create sample HL7 messages, you may have to validate the created Message object by using ParseMessage. For example, if you try to use the 3rd test case code in the HL7 Message Generator for testing purposes, you have to call ParseMessage before adding it to the *messages* variable. But what we do here is generate test message regardless of their validity and verify the results after message transmission.

Finalizer

Since the purpose of this service is to generate the test messages and send to the ORM Sender, we don't have any specific work after the Steps. So, keep this part empty.

Conduct a test

Deploy services

Since we have finished the service tester, deploy it to the server. In addition, in the previous sections, the ORM Listener was deployed for demo purposes. But we haven't deployed that service, so we need to deploy that as well. You can load the services one by one or select the project node to load them all. But please don't forget to verify ( ) each service before loading it. Once the services are loaded, **switch to the Management workstation**.

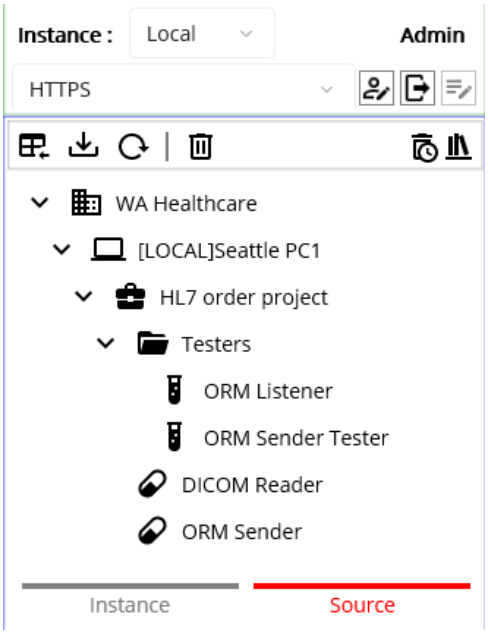


Figure 175.Loaded services on the Macaron server

When you click the Source tab, you can see loaded services on the Macaron server, as shown in Figure 175. We will not deploy the DICOM Reader because the ORM Sender Tester will do that role instead. But we can deploy all and undeploy the DICOM Reader later. Select the domain node, which is WA Healthcare, and click the Deploy button.

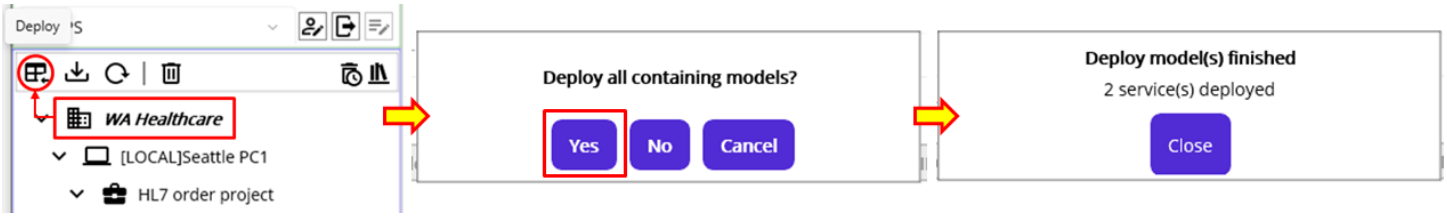


Figure 176.Deploy the domain model from Source tab

When you see a popup that asks you to deploy all, click Yes button. And you will see the result messages that 2 services were deployed, as shown in Figure 176. As you may notice, the services under the Testers folder will not be deployed as a group, not to mention they will not be auto-deployed when the server starts. Click the Instance tab to see the deployed services.

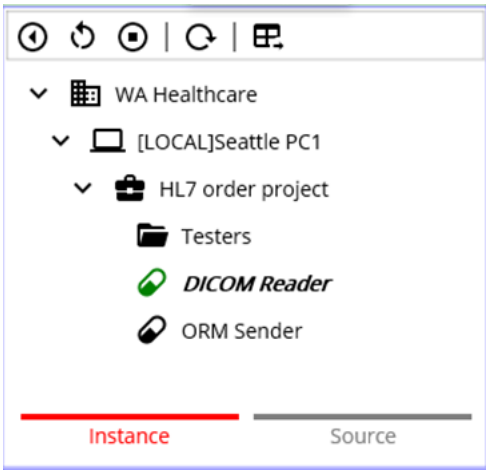


Figure 177.Deployed services

As you see the services in Figure 177, only the DICOM Reader was started. As a side note, the HL7Client will not start if the target destination is not available because it is connection based service by design. But the DICOM Reader that sends the messages to the

ORM Sender was started. If the ORM Sender is deployed, whether it is started or stopped, **the DICOM Reader will keep enqueuing the messages to the ORM Sender**. Once the ORM Sender starts, i.e., the target application is available, dequeued messages will be processed. On the contrary, if the ORM Sender is not deployed, deployed DICOM Reader starts but will keep producing error message, as shown in Figure 178.

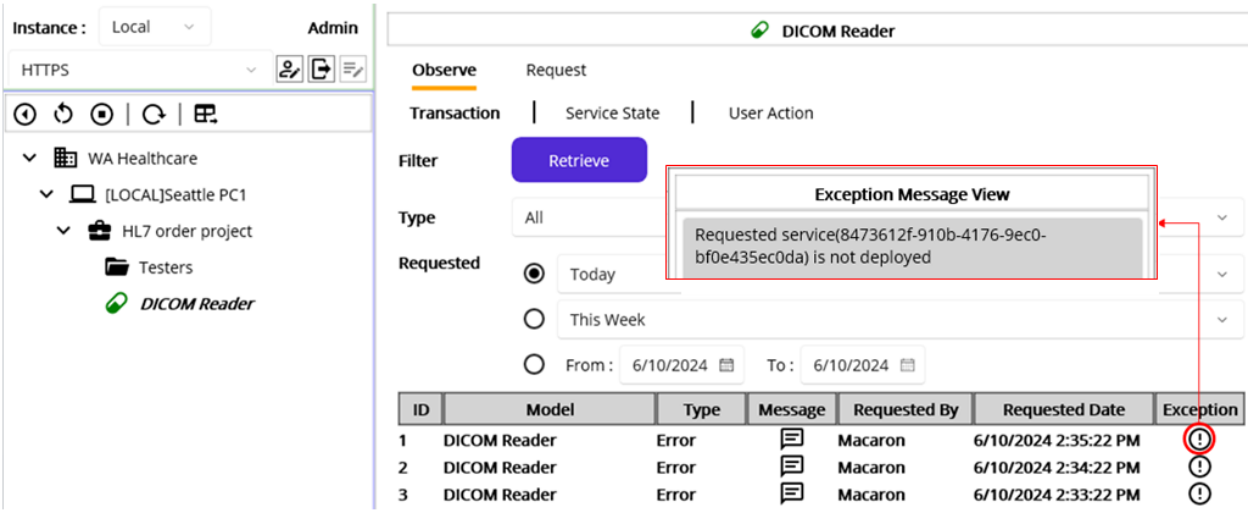


Figure 178.Mapped service is missing case.

As we don't need the DICOM Reader service, we can undeploy the service. But at this time, we know that the ORM Sender has messages enqueued by the DICOM Reader. So, undeploy the ORM Sender as well to make sure we don't have any data from the DICOM Reader.

Flush the enqueued messages in the HL7Client

Since the HL7Client service has a queue to process, it is open to the other service to receive any message as long as the HL7Client is deployed to support later processing when the service starts. But when the service is stopped and you want to flush the queue, you can choose either to deploy it again or to click the Restart ( ) button. That will create a new instance of the HL7Client service and will have an empty queue as a result. You may not need this frequently, but you can use it while testing the services.

Undeploy Services

We don't need the DICOM Reader to test the ORM Sender and can undeploy. And we need to flush the queue in the stopped ORM Sender as well. Instead of restarting the ORM Sender, we can keep it as is and redeploy it later, which replaces it.

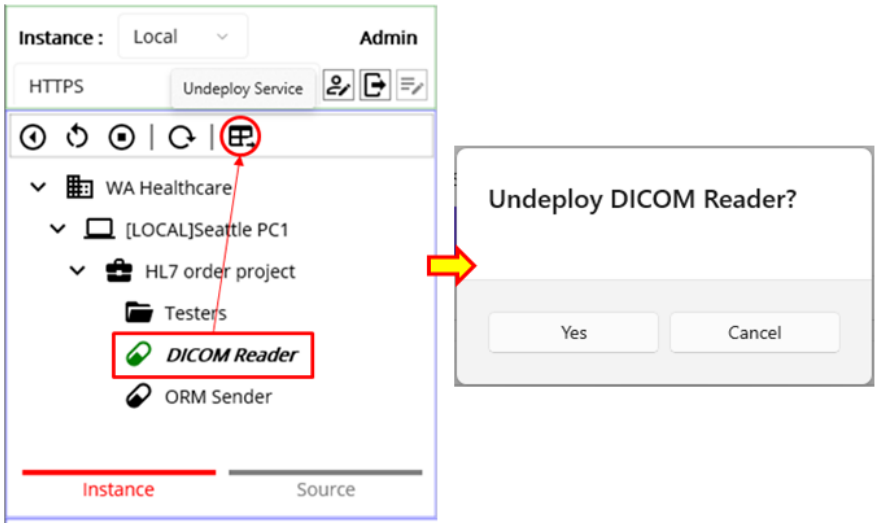


Figure 179.Undeploy the DICOM Reader services

Select the DICOM Reader node and click Undeploy ( ) button. Once the popup asks you to undeploy the service, click Yes button, as shown in Figure 179. Then we can go back to Source tab.

Service Dependencies

Before we redeploy the services, let's take a look at the service dependencies. Since we created the ORM Listener that acts as a client's application in the Testers folder, the service will not be deployed when we select a group model, such as domain or project, and click the Deploy button. And the ORM Sender existing outside the Testers will be deployed but not started because of the destination is unavailable. See following Table 27 for the service dependencies.

Service types	Example services	When deployed	Request Map target
CodeRepeater	DICOM Reader	Starts automatically	HL7Client
HL7Client	ORM Sender	Starts when HL7Listener is available	HL7Client
HL7Listener	ORM Listener	Starts automatically	HL7Client

Table 27.Service Dependencies

The first thing to note is the HL7Client, which is the ORM Sender that highly depends on the HL7Listener or external application. So, when you deployed the service, please make sure the target destination is available or the HL7Listener is deployed and running. Especially when you put the HL7Listener service in the Testers service, you can face an unexpected result when the server restarts that doesn't deploy the services in the Testers service. So the required services pointing at that service will be stopped as a result. In addition, the CodeRepeater that sends the message to the HL7Client service will keep sending message and the queue in the HL7Client could grow enormously. This will not be a serious case because this will happen mostly in the local testing with your own machine that checks frequently. But please take note this one while you testing services.

The second thing is the Request Map that each service can call. Currently, Maca doesn't provide the RequestListener that listens to a message. So, only the HL7Client that has a message queue supports the Request Mapping. You can revisit Figure 142 and 143 for the Request Mapping. This makes a service send a request containing a message to the target service, but the message will be lost when the target service is deleted. In addition, if you copy or clone the service that uses Request Mapping, you need to find the location and modify each Request call one by one to a new service. So, please take note this one as well while you testing services.

Redeploy Services

As we saw that the ORM Sender requires the ORM Listener, we can deploy the ORM Sender first, as shown in Figure 180.

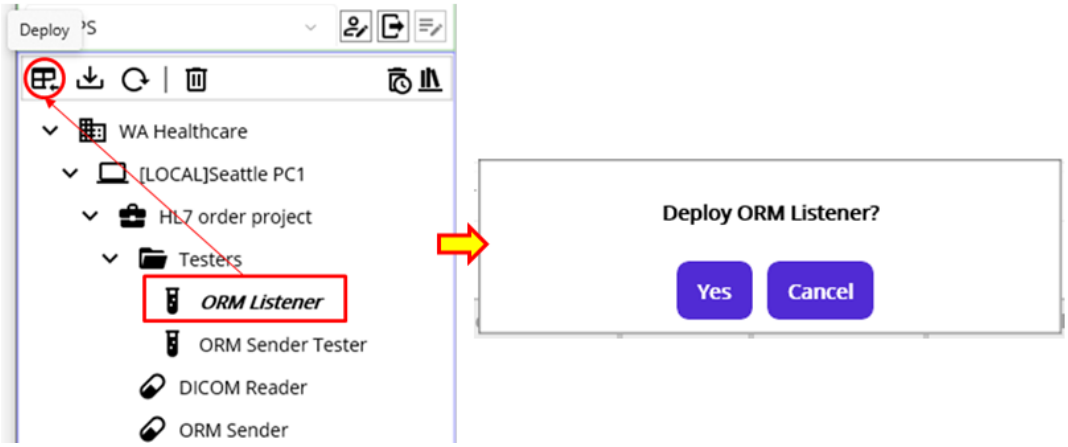


Figure 180.Deploy the ORM Listener first

After that, deploy the ORM Sender. Lastly, deploy the ORM Sender Tester. Once you finished the deployment, **click the Instance tab** to see deployed services.

Retrieve the transaction

Retrieve ORM Sender transaction

As we set the test messages to be sent to the ORM Sender, we can select the ORM Sender node from the model explorer to retrieve the results because the ORM Sender Tester was started when it was deployed. Select the ORM Sender node. Then pick Today for Requested and click Retrieve button, as shown in Figure 181.

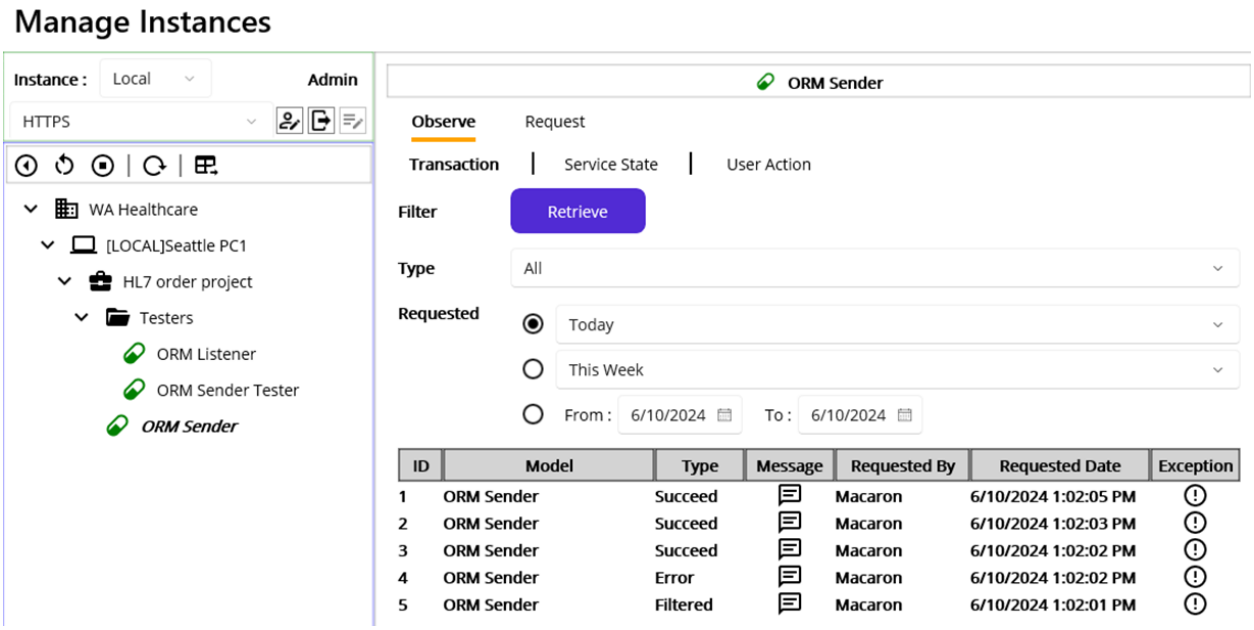


Figure 181.The result of ORM Sender transaction retrieval

As we sent five test messages, we got five results with different transaction types: Succeed, Error, and Filtered. Although we can let the ORM Sender Tester keep going, we can stop the service using the Stop ( ) button to prevent the records from piling up. Now, take a look at the result one by one.

As you may notice, the records are displayed in descending order, so the 1st message would be the 5th record. Since the transaction was Filtered as expected, click the Message ( ) and the Exception ( ) buttons to see the exchanged message(s) along with any issues occurred

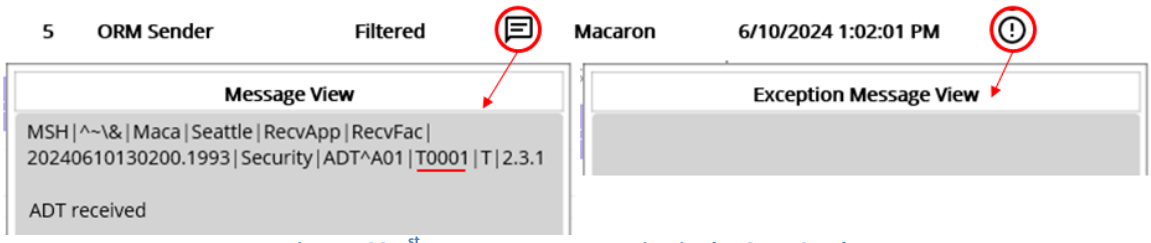


Figure 182.1st test message transaction in the ORM Sender

The record is the result of the test case 1 in Table 24. Since the message was filtered by the ORM Sender because we sent an ADT message, we don't have any returned ACK and thus show only generated message, as shown in Figure 182. So, this message with T0001 shouldn't be in the ORM Listener that we will retrieve later. We don't have any special point to check here, so, move on to the 2nd message.

Note: As mentioned earlier, Maca proto doesn't save the Filtered reason to the database. And the Exception Message View shows only error or exception information. So, we don't know what made the message filtered at this point other than Filtered flag. However, you can make the Filtered transaction an Error in the Filter or Steps to show the reason if you want, if that makes the service more controllable.

Click 4th record's Message and Exception to see the result.

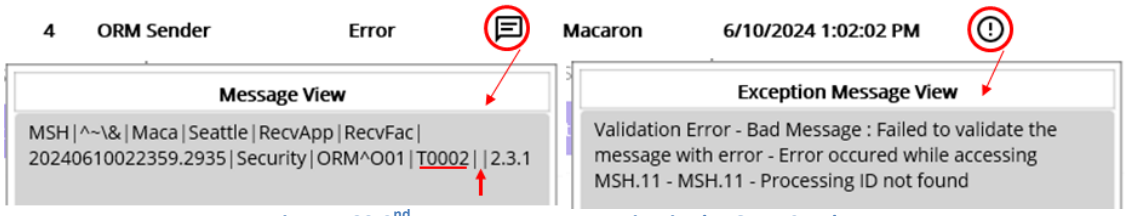


Figure 183.2nd test message transaction in the ORM Sender

This record is the result of test case 2 in Table 24. As you see in Figure 183, the message was not sent because the required field MSH.11 was missing. You can see the detail of the issue in Exception Message View, as shown in Figure 183. Click 3rd record's Message and Exception to see the result.

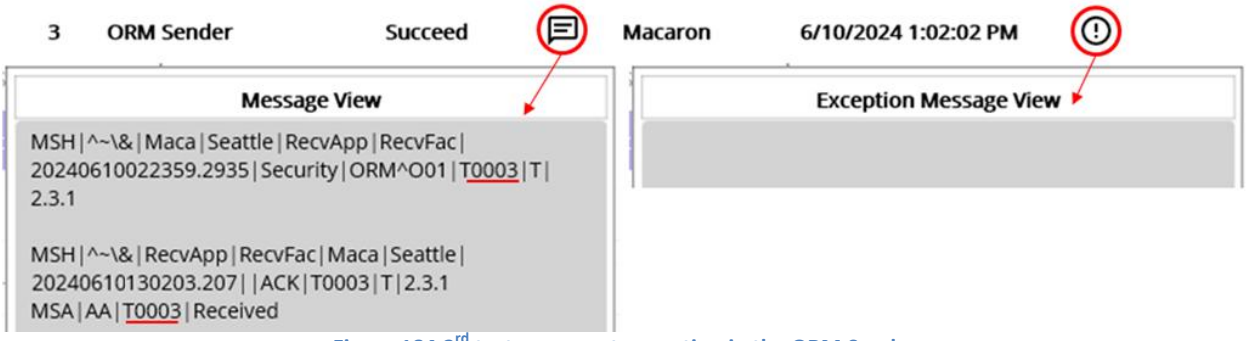


Figure 184.3rd test message transaction in the ORM Sender

This record is the result of test case 3 in Table 24. Unlike the first 2 test cases, the Message View shows the returned ACK message below the sent message, as shown in Figure 184. As we expected, the returned ACK has AA in MSS.1 and Received in MSA.3. When you click Exception button, you will see no issue happening in either ORM Sender or ORM Sender Tester. However, we set the ORM Listener to send always the fixed ACK, there might be an error in the ORM Listener side. We will send the record in the ORM Listener later. But for now, we can move on to the next record. Click 2nd record's Message and Exception buttons.

Note: Unlike normal HL7 messages, Maca will save the whole ACK message to the embedded database. Because the ACK is a result of the message processing, it used to contain pre-defined data along with any issues found while processing. So, you can track the message using this returned ACK message.

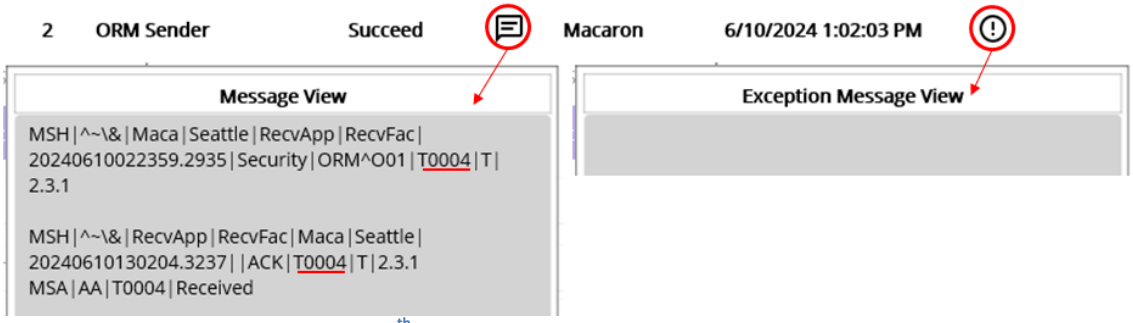


Figure 185.4th test message transaction in the ORM Sender

This record is the result of test case 4 in Table 24. As you see in Figure 185, this is similar pattern with the 3rd record, which is the test case 3. We have a returned ACK message with fixed values and no issue found in the Exception Message View. Like the 3rd record, we need to check the ORM Listener side as well to check what happened. But at this point, this is also as expected. Now, move on to the last record. Click the 1st record's Message and Exception buttons.

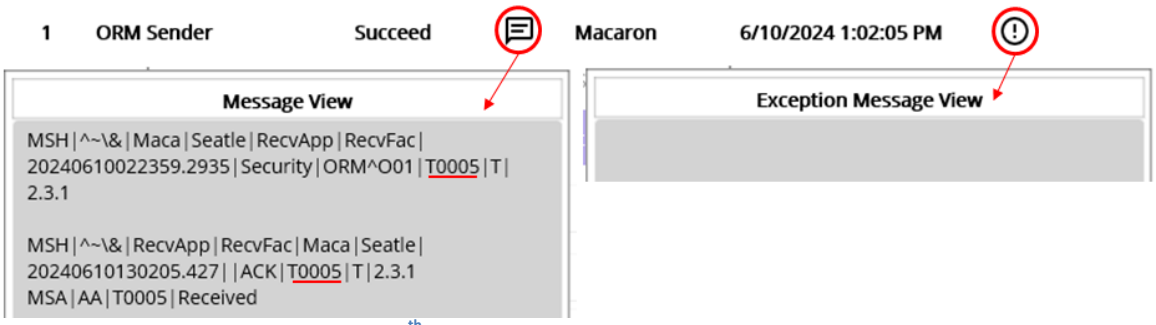


Figure 186.5th test message transaction in the ORM Sender

This record is the result of test case 5 in Table 24. Like the 3rd and 4th records, there is nothing special to check here. So, let's move on to the ORM Listener that processed the ORM and returned the above ACK messages. And we can cross-check the results.

Retrieve ORM Listener transaction

From the model explorer, select the ORM Sender node under the Testers. Since we are still in Observe > Transaction view, keep the options as is and click Retrieve button.

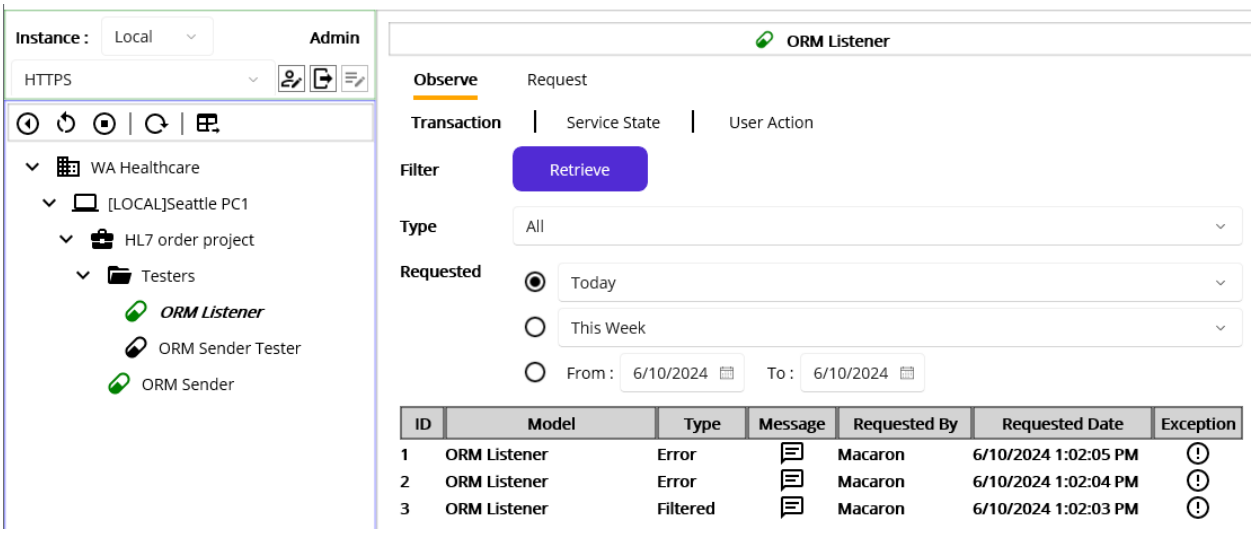


Figure 187.The result of ORM Listener transaction retrieval

As you see in Figure 187, there are 3 records. As you may already notice, the ORM Sender sent only 3 messages after filtering test case 1 and 2. But, unlike the results in the ORM Sender, we can see two records are marked Error. Let's take a look at each record. Click the 3rd record's Message and Exception buttons, which is the 3rd test case.

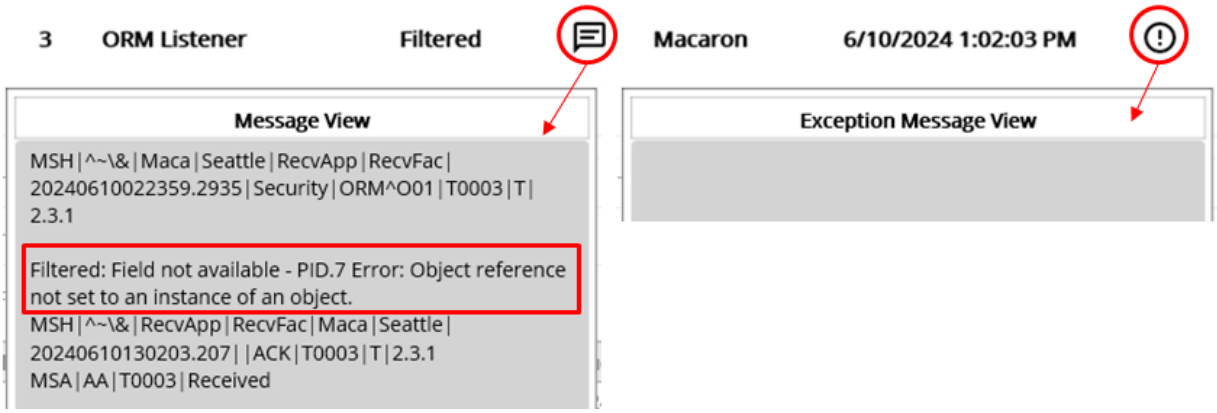


Figure 188.3rd test message transaction in the ORM Listener

As you see in Figure 188, the Message View shows the received message at the top and corresponding result ACK along with the error message added to the ResultMessage property. You can see the error message above the ACK is Filtered. Based on that, we assume that PID.7 accessing was failed and the transaction was marked as Filtered followed by the code in Steps. When you look at the code in Figure 170, there is a line that checks PID.7 field. If you look at test message #3 in Figure 174, you will notice that PID.7 doesn't exist, like the below underline in red.

```
|ORM^O01|" + testID + "|T|2.3.1\rPID||12345|||DOE^JOHN\r";
```

So, message.GetValue("PID.7") in Figure 180 threw HL7Exception and thus the transaction was marked as Filtered in the Steps. You can check Table 25 again for this issue. Since the exception was handled by the code, we don't have any error message in the Exception view. Based on this, the receiving end knows the issue but the sending end doesn't know about it. So, you may think that this is not ideal in terms of message troubleshooting. However, this is a hypothetical scenario that might happen, so we can use this scenario for practice purposes.

Now, let's look at the 4th test case. Click the Message and Exception buttons in the 2nd record.

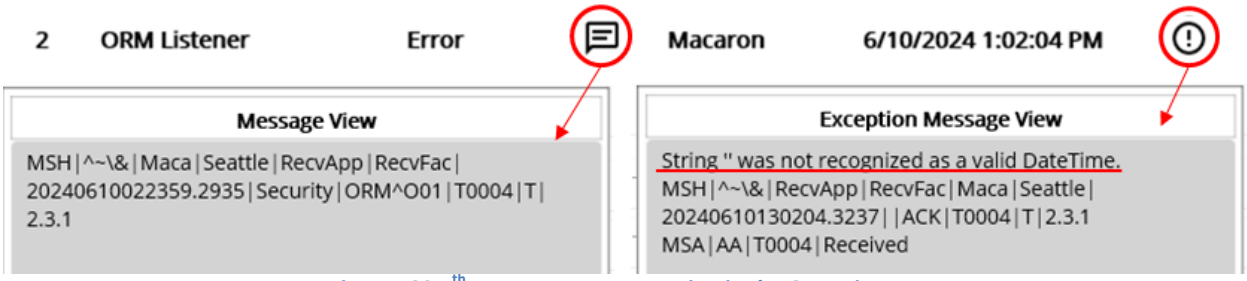


Figure 189.4th test message transaction in the ORM Listener

As you see in Figure 189, there was an error with 4th test message. If you see the Exception Message View, there is an error message above the ACK message. It is not clear what exactly made that error for now. As a shortcut, this is thrown by DateTime.Parse method. It is related to the empty value returned by the message.GetValue("PID.7") out of the following message in test case 4 in Figure 174.

```
|ORM^O01|" + testID + "|T|2.3.1\rPID||12345|||DOE^JOHN||\r";
```

Because DateTime.Parse[34] strictly requires the date time value, it throws the FormatException when the parameter is not date time format. Since the PID.7 value is empty string, DateTime.Parse(message.GetValue("PID.7")) was interpreted as DateTime.Parse("") and it throws the FormatException as a result. And the exception caught in Exception block and thus the transaction was marked as Error. Like this exception case, debugging is sometime quite hard in Maca unless you set another layer of checking, such as TryParse[35] method that returns Boolean value depends on the input parameter. But before calling that method, it is always recommended to check the value in the field is empty or null first so that you can decide to proceed with Parse method or not.

Lastly, let's check the last test case. Click the Message and Exception buttons in the 1st record.

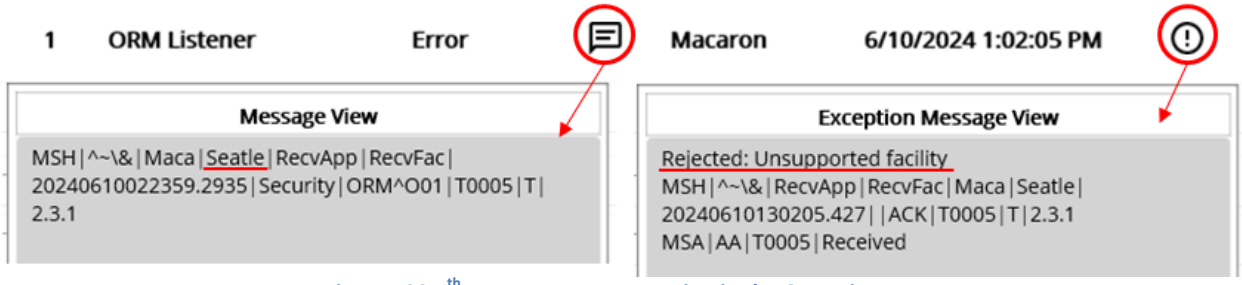


Figure 190.5th test message transaction in the ORM Listener

[34] https://learn.microsoft.com/en-us/dotnet/api/system.datetime.parse?view=net-8.0
[35] https://learn.microsoft.com/en-us/dotnet/api/system.datetime.tryparse?view=net-8.0

Before you look at the Exception Message View, you will probably be surprised by the 1st record marked with Error instead of Succeed. As we provided all required values, there should be no issue with the message itself but somehow the message was rejected. If you look at the Message View, you will be realized that the sending facility was incorrectly typed as Seatle instead of Seattle. Based on the switch statement in Figure 170, the value didn't fall into either Seattle or Bellevue cases. So, the default case was processed instead, and the transaction was marked as Error as a result. While handling string values, typos are a typical error that we will face. There could be a couple of ways to handle this issue, such as using enums or public readonly variables. To facilitate the development and prevent typos, Maca provides extra variables named Service References.

Request a Service

As you remember the Request tab that we briefly mentioned in the "Request Instance Model" section, you can type the custom HL7 message to test the ORM Sender service as the ORM Sender Tester service does. If you look at the Figure 104, the ORM Sender listens Damia's request, specifically the Admin's request, and thus we can try that feature in this step. You can select the ORM Sender service and click the Request tab. Then the Request UI will show up in the Instance model viewer, as shown in Figure 191.

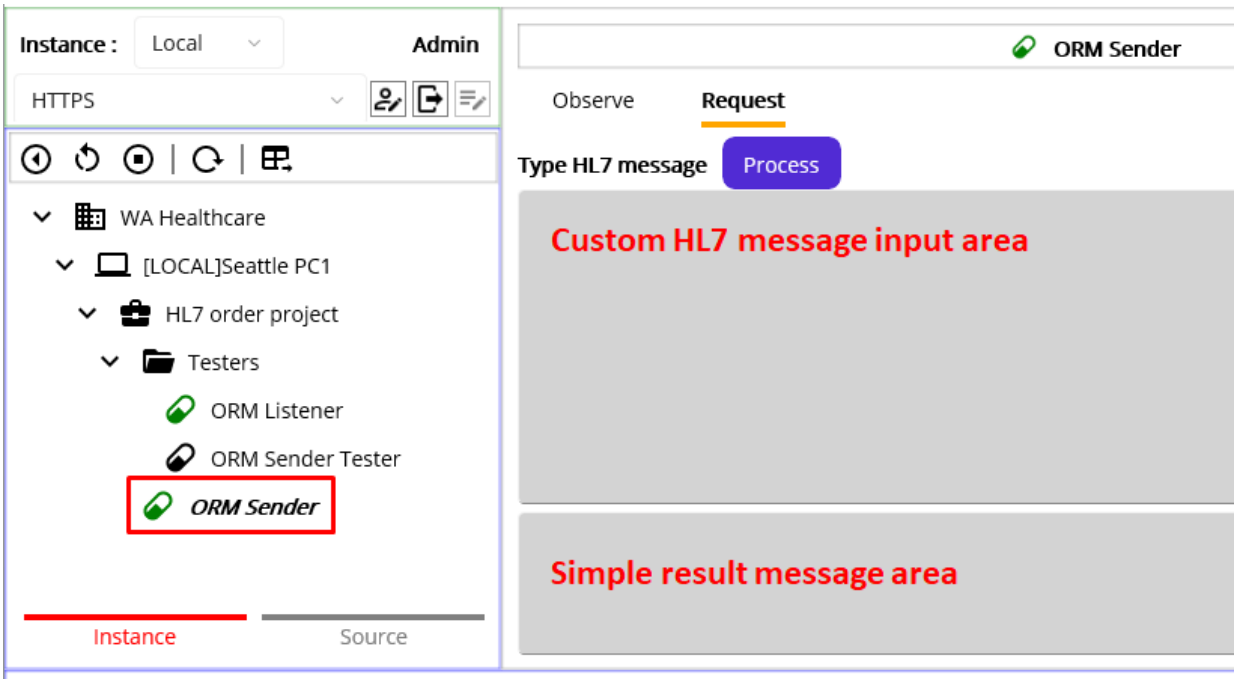


Figure 191.Request UI for HL7Client service

The UI is composed of two simple main areas: a custom HL7 message input area and a result message area. You can type any kind of HL7 message or paste the message copied from any resource for your test. For this tutorial, we are not going to touch all test cases because you can try them by yourself. Instead, we will focus on how this feature actually impact the transaction in terms of user action history.

As you see in Figure 191, we logged in as the Admin account, the built-in default user based on Table 3. This means any HL7 message typed in the message input area will be tagged as "Admin" when the Process button is pressed. To see the difference, let's type a sample message, or copy below message in yellow box, as shown in Figure 192.

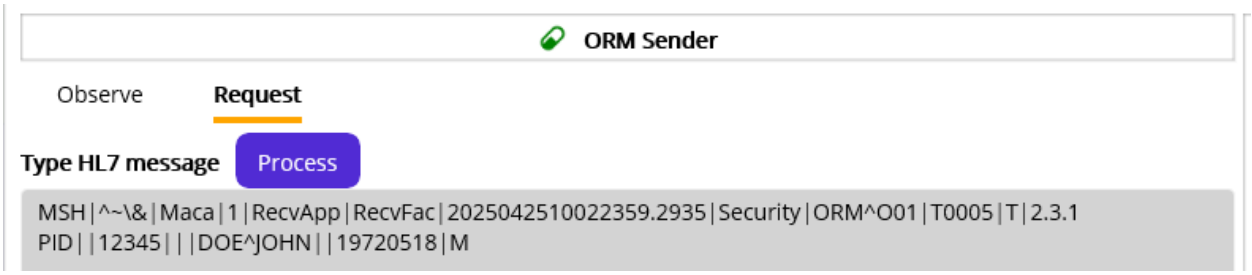


Figure 192.Custom HL7 message request

```
MSH|^~\&|Maca|1|RecvApp|RecvFac|2025042510022359.2935|Security|ORM^O01|T0005|T|2.3.1
PID||12345|||DOE^JOHN||19720518|M
```

Click the Process button to request typed in custome ORM message. Then go to **Observe > Transaction** tab. Select "**Today**" for the Requested option and click the Retrieve button, as shown in Figure 193.

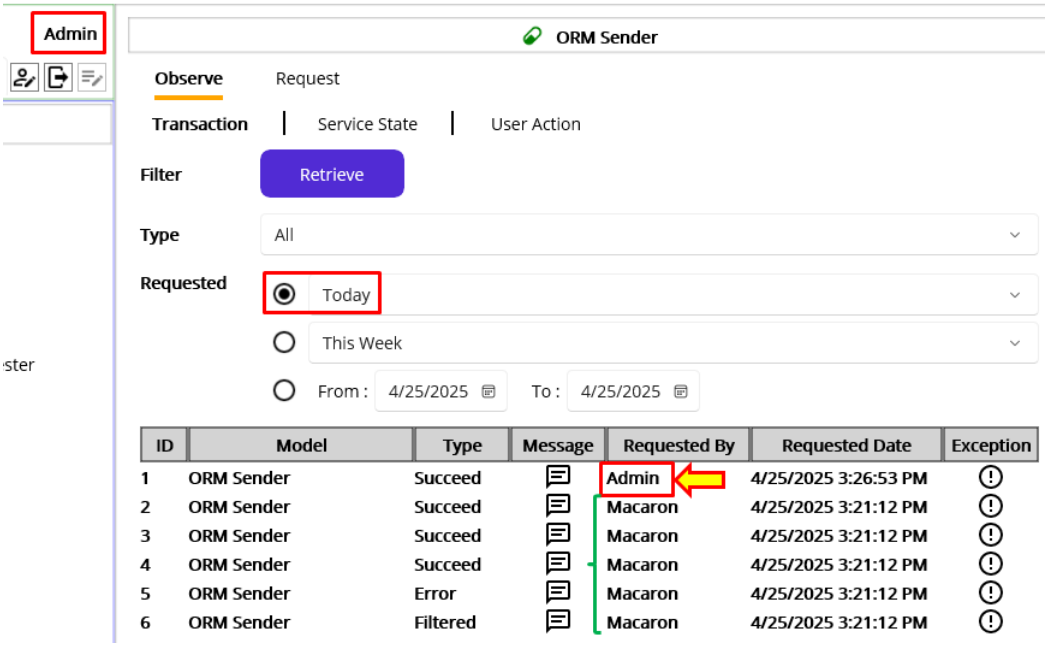


Figure 193. A transaction made by Admin user

As you can easily notice, the transaction ID 1 is made by the user, Admin. And all the previous transactions were marked as "Macaron" followed by the "Requested By" column information, which means Macaron managed the transactions. For now, Maca doesn't implement role-based user restriction. So, there is no feature to prevent a logged-in user to send custom HL7 message to the destination application. However, even if a high-level user is logged in, he/she can still send HL7 messages at any time. Instead of focusing on the priviledges, Maca tracks who sent those messages.

Now, let's try the same message with a different account. For the testing purpose, the new user "**sidekick**" was created and logged in. You can click the "Sign Up" button on the popup window to register (create) an account after you click the login ( ) button. By using the same HL7 message in Figure 192, we clicked the Process button to request the ORM Sender service. Then go to **Observe > Transaction** tab. Select "**Today**" for the Requested option and click the Retrieve button, as shown in Figure 194.

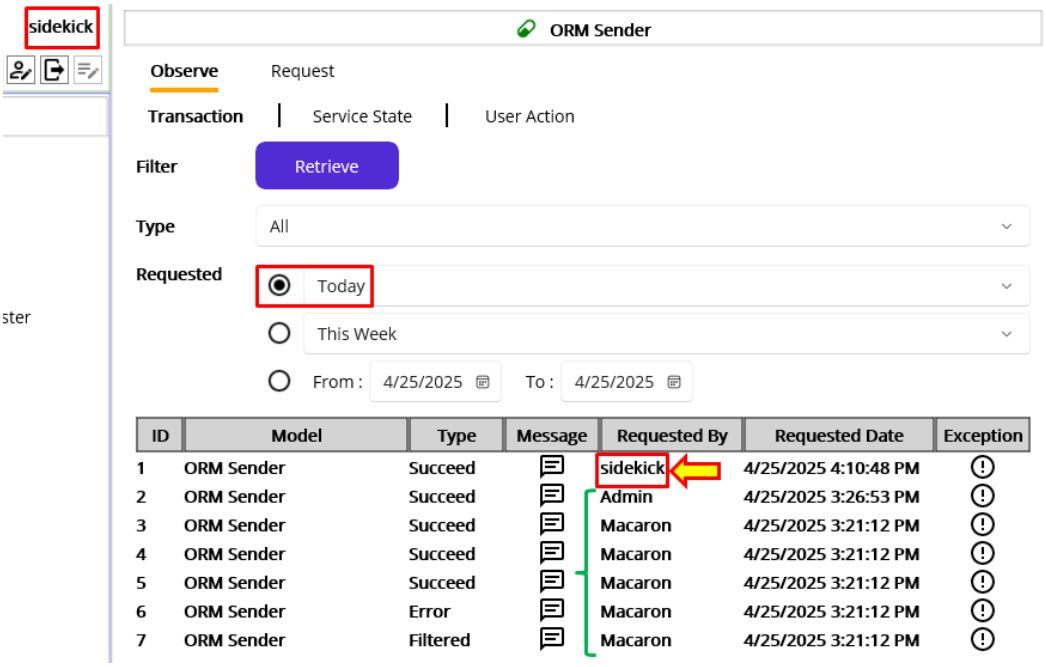


Figure 194. A transaction made by sidekick user

As the results clearly shows, we can identify the sidekick user sent a custom message to the ORM Sender service. Although this is quite helpful to identify who requested the service, it is still not enough to track the user activities. For example, a user can request previously processed messages, i.e., reprocessing. Or, a user may see the sensitive messages using message button. To track all user-driven activities, just checking the Requested By column provides limited information.

To support the user-driven activity history, **next release** of Maca will provide User Action Observation feature, as shown in Figure 195.

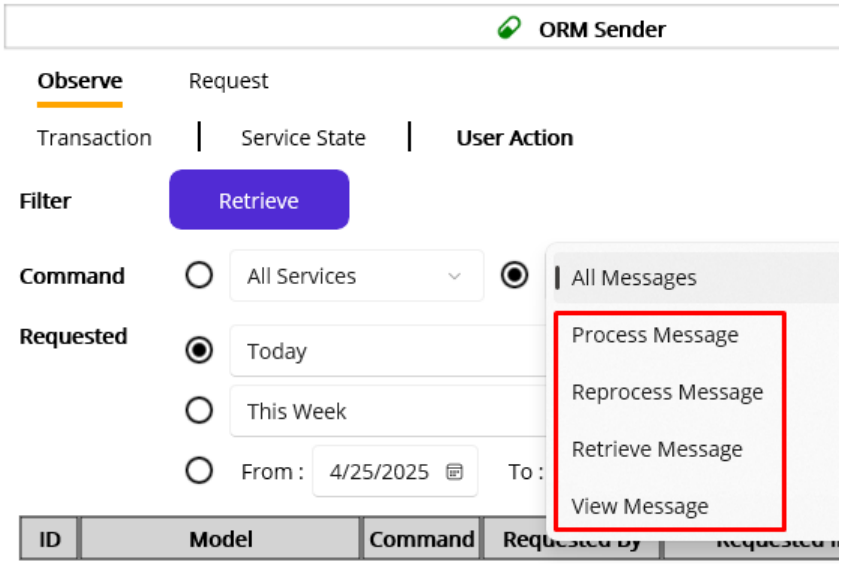


Figure 195.Message related user action observation(***Not implemented***)

In the Command filter, there is a picker that allows to choose Message related user action. Please check the below Table 28 for the detail.

Message Commands	User Actions	Results
Process Message	Click "**Process**" button on the Request tab	Transactions made by the users
Reprocess Message	Click "**Reprocess**" button on the Message View popup	Transactioins made by the users

Retrieve Message	Click "**Retrieve**" button on the Observe tab	Retrieval history of the users
View Message	Click message ( ) button on the results.	Message view(open) history of the users

Table 28.Message commands details (***Not implemented***)

Note. The UI, including the retrieval options, is not fixed to support this feature. So, there would be some UI changes in the next release.

Track Deactivated Services

Since Maca allows any registered user to log in to the Macaron server, it is still important to track ***who delete*** the services, either accidentally or on purpose. In addition, it will be helpful to track ***who loaded*** the services to the server or ***overwrite*** the existing service. In the previous section, we saw that who requested the requestable services by checking the Requested By column. But that is for the deployed services, especially for the transactions, the service deletion is not retrievable in the Instance tab. Since the deletion activity is related to the physical files, it is more relevant to **the Source tab**. Click the Source tab, then see the below Figure 196.

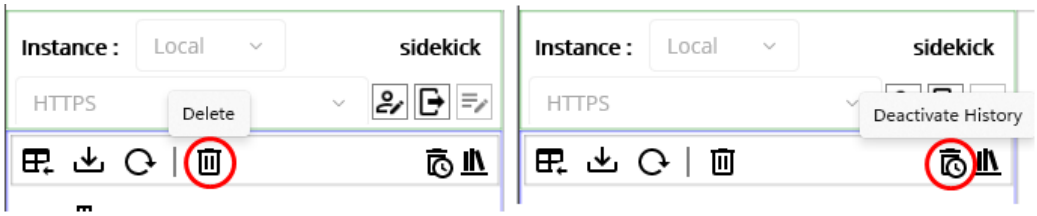


Figure 196.Delete activity related buttons on the source tab

You can see the Delete button and the Deactivate Histor button in the above Figure. The delete button triggers the deleted file information to be saved in the database. As you may notice, the model can be frequently loaded and deleted to make sure file update or permanent deletion. So, instead of the delete history, we call it "**Deactivate**" history. Please click the Deactivate History button, as shown in the Figure 197.

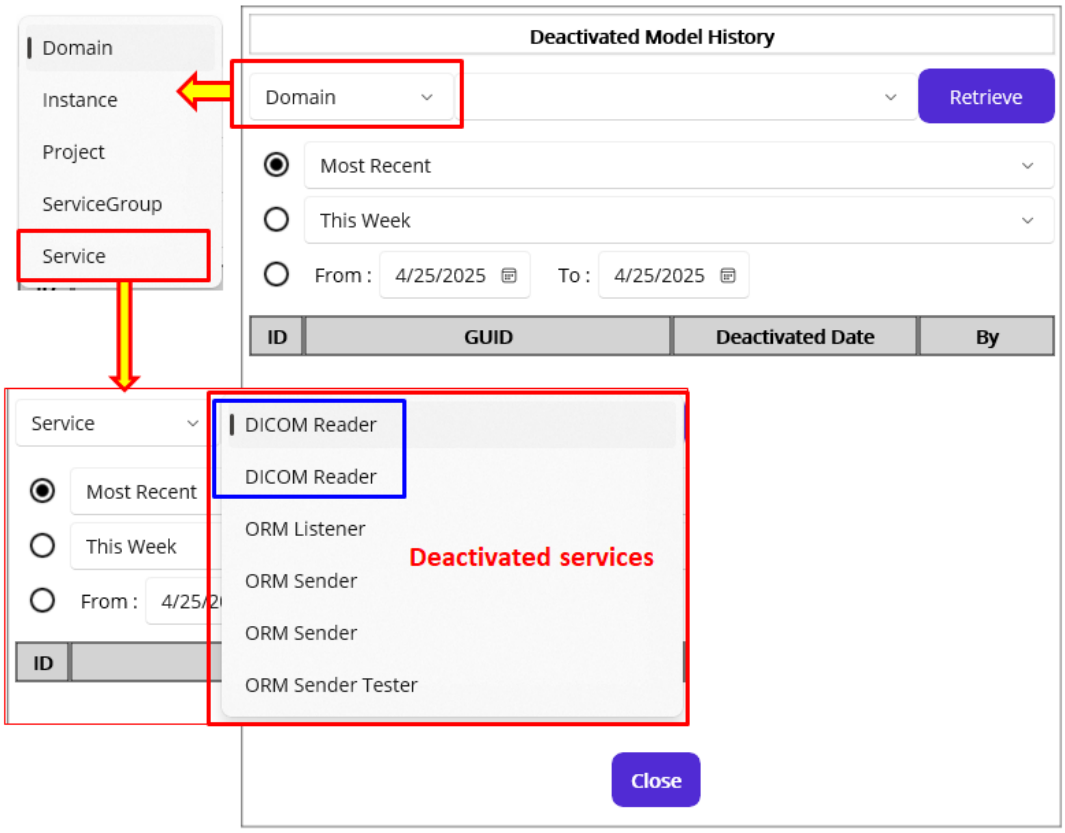


Figure 197.Deactivated Model HIstory popup

In the above figure, we selected “Service” model first from the model types, then the deactivated model picker shows all previously loaded and deleted services. And you may wonder why we do have 2 identical “DICOM Reader” in the picker. The first guess, which is not common, is that a user created one and deployed. Then he/she physically deleted the service in the Management workstation and the Develop workstation. Then a user created a new one with the same name and then deployed the service.

The second guess, which will be common, is the user created the service in a different location. By design, Maca doesn’t allow identical service name. But that doesn’t mean it restricts the identical service name in the other group or project. In other words, we can have the DICOM Reader service under the Project and Testers group at the same time, as shown in the Figure 198.

Figure 198. The same DICOM Reader services in a different location

Those services with an identical name, probably on purpose for the testing purpose, may behave exactly the same or differently based on the defined code. However, when you create or clone a service or add a service from the template, the new GUID will be assigned. So, the two services will not have the identical GUID. And we can assume that two DICOM Reader services in the deleted service picker will have a different GUID. To demonstrate the case, Figure 199 shows the two DICOM Readers history comparison.

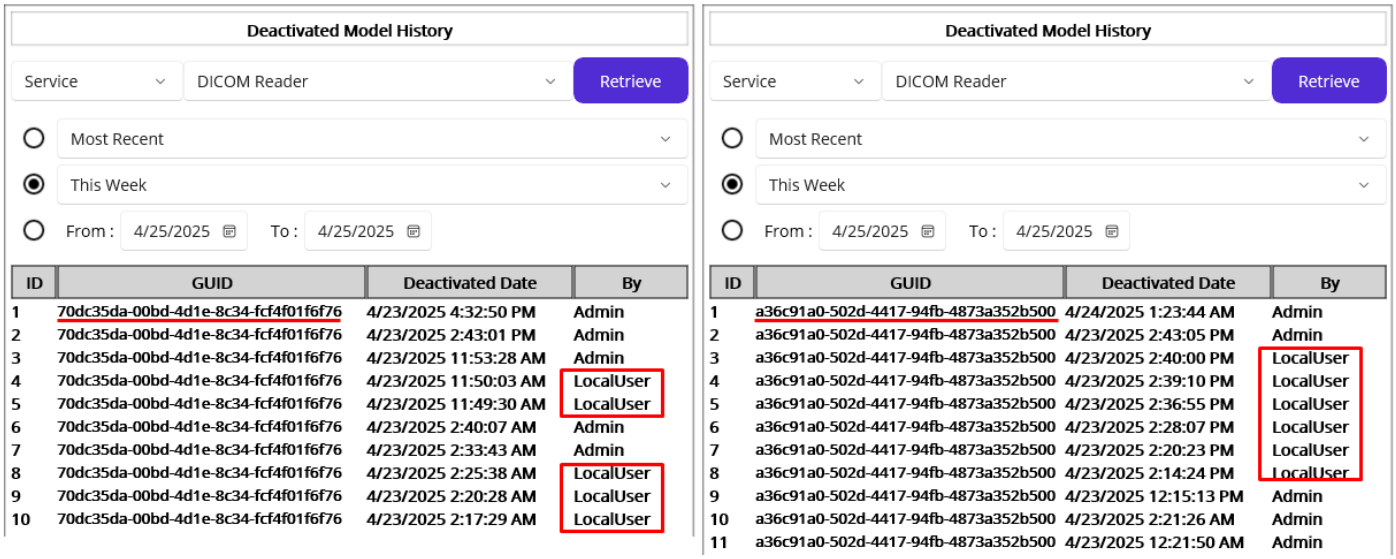


Figure 199.Two DICOM Reader services history

As you see, each GUID is different. So, we can assume that the two services were loaded into the different project or group. Or, the first one deleted from the server and then a new service with a different GUID was loaded. Still there is a possibility that no such file exists on the server. But if it exists, we can check the service’s GUID in the Development workstation where you keep the latest version to see the server side service is correct one.

Another thing you can check is the “By” column. The red box shows that the service was loaded by the LocalUser, i.e., the service was loaded from the Development workstation using the Load ( ) button. And the Admin may load ( )the service or delete the service from the Management workstation.

To give you an example, we can delete the ORM Sender Tester service. As a reminder, currently logged in user is “sidekick,” which we just created. Select the ORM Sender Tester service from the model explorer and then delete the service. (If you see the popup that asks you to undeploy the service first, then go to the Instance tab. Then undeploy it.) Then we can retrieve again with the service, as shown in Figure 200.

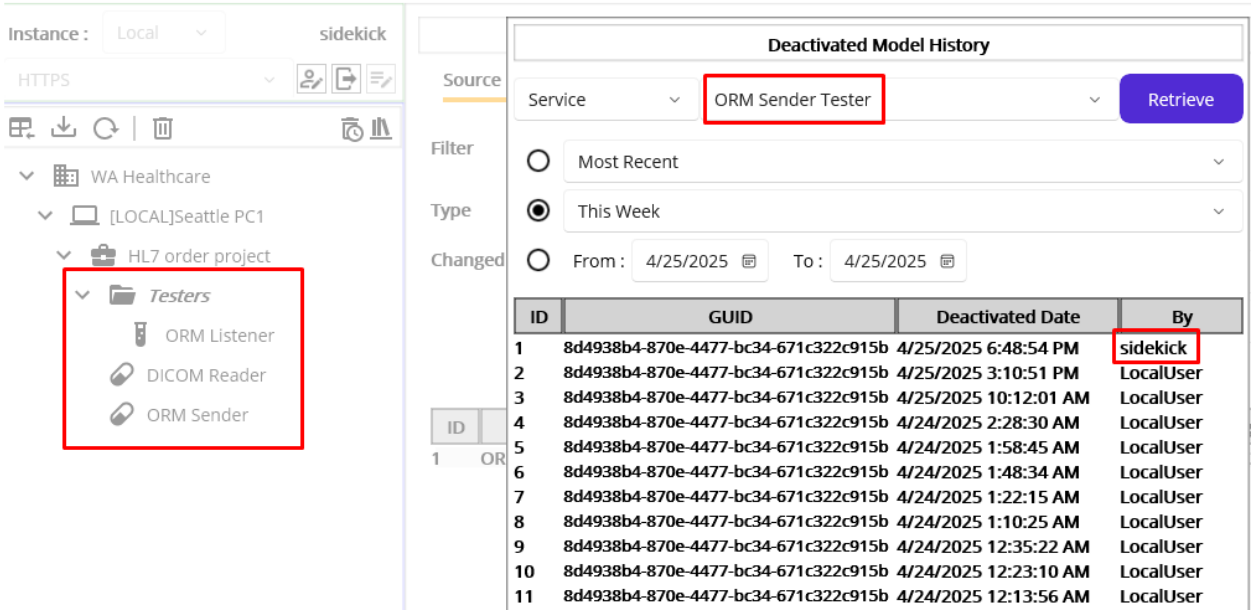


Figure 200.Deleted history by sidekick

As you see on the model explorer in the Source tab, there is no ORM Sender Tester service, i.e., no physical file on the Macaron server. But we still can keep track of previously loaded model’s history and can see the last user who deleted that file.

By using the Deactivated Model History UI, you can check when the service was loaded, overwritten, or deleted along with the related user who committed that action.

Service References

Like the 5th test case that was incorrectly typed, we can face such typo issues in many cases. Especially the key terms that can be used repeatedly in many locations, finding typos and fixing them could be a hard mission in a lengthy code. In addition, if the value needs to be changed after the system upgrade or design change, we have to revisit the service and do the error-prone job again. Like the location values, such as Seattle and Bellevue, we may set those key terms as Service References and just make the services point at them.

Basic concept

If you recall, Maca was built with hierarchical models. Each group model can have its own variables that can be easily identifiable by its model type and are hierarchically accessible. To illustrate this concept, we can revisit the 5th test case that has a typo issue. For example, MSH.4, which is a sending facility, was set as Seattle in the test cases. Based on the hypothetical scenario, you are working at the Seattle location, and the machine named “Seattle PC1” could be considered a facility that sends the ORM messages. So, instead of hard-coding the sending facility value in the service file, we can set the value in the Instance model and let the service reference it. Following Figure 201 will describe visually for the DICOM Reader case.

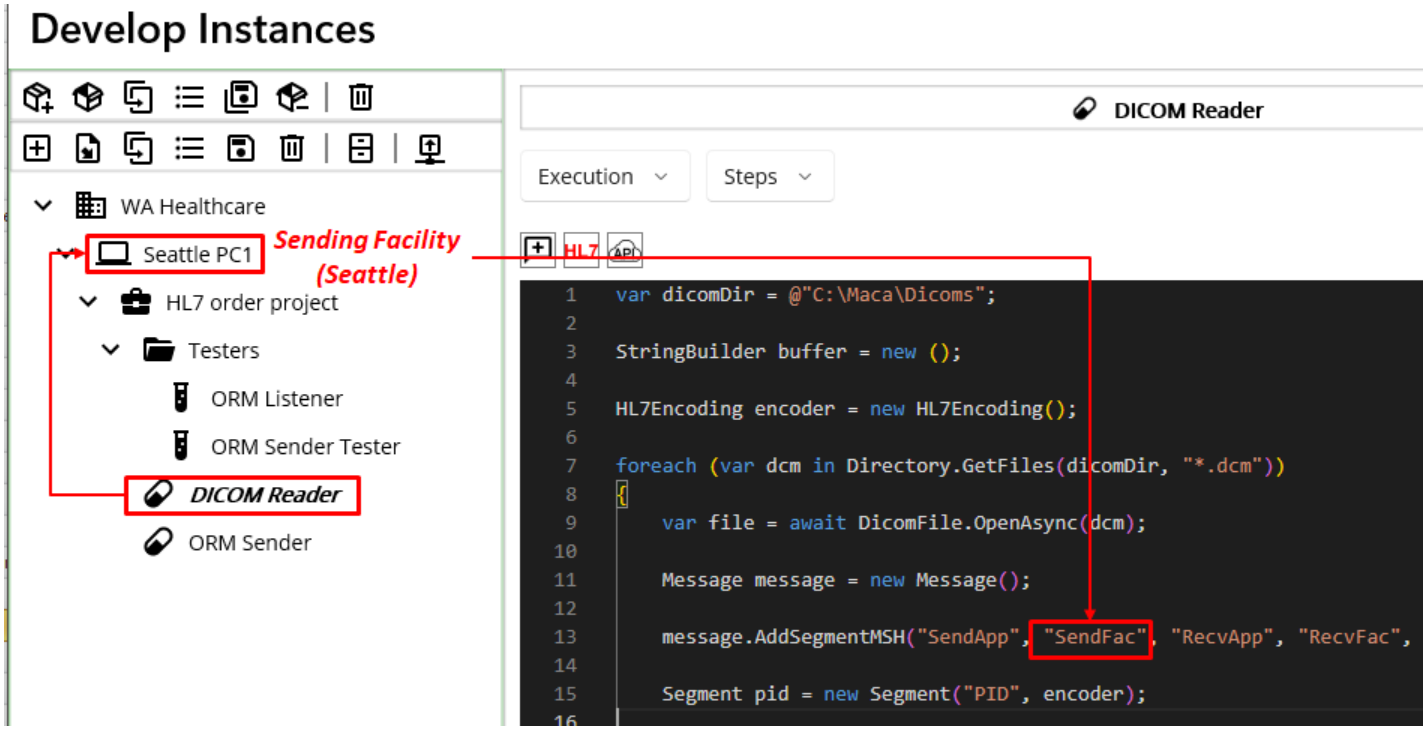


Figure 201.Service References example

As you see Figure 201, the flow shows what you can do with the Service References. The first thing to do is looking at the Instance to get the Sending Facility value, i.e., Seattle. And then populate the value in the 2nd argument of the AddSegmentMSH method. But the question at this point is how to set the Service References and how to use them. So, let's look at how to set the References, first.

Set References

Since we are going to set the Instance References, select the Seattle PC1 node. Then you will see the Detail view of the Instance model, as shown in Figure 202.

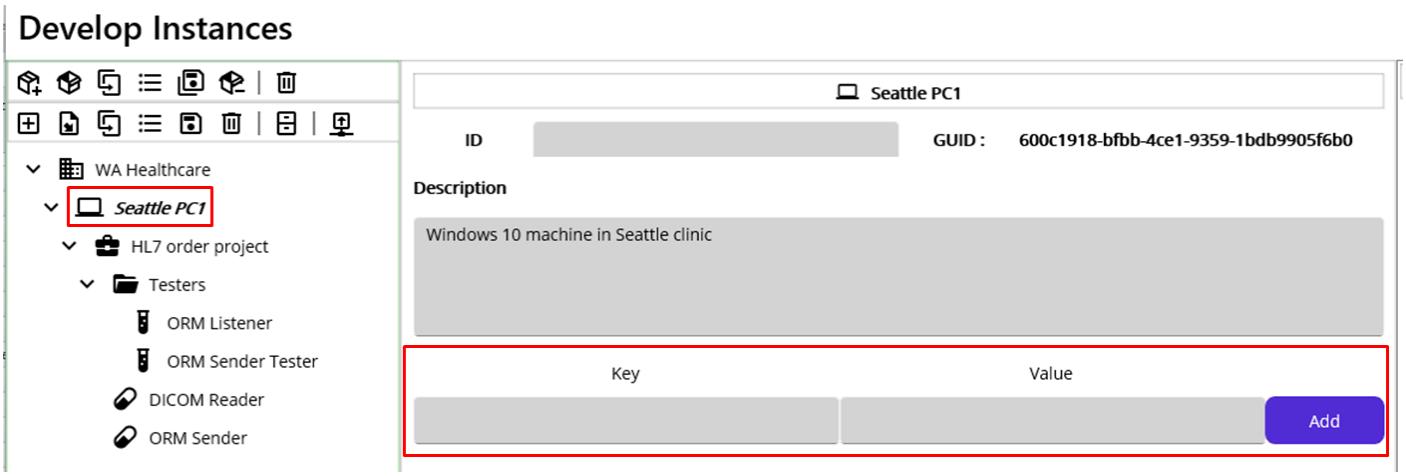


Figure 202.Instance References

In order to use the References, we have to put the values in the target group model, in this case Seattle PC1. When you select any of group models from the project explorer, you will see the detail along with Reference input area, as shown in Figure 202 with a red box. So, what you can do here is typing Key and Value that can be added to the Instance. As we need the sending facility, we may type SendingFacility for Key and Seattle for Value and click the Add button. Then you will see the added Reference, as shown in Figure 203.

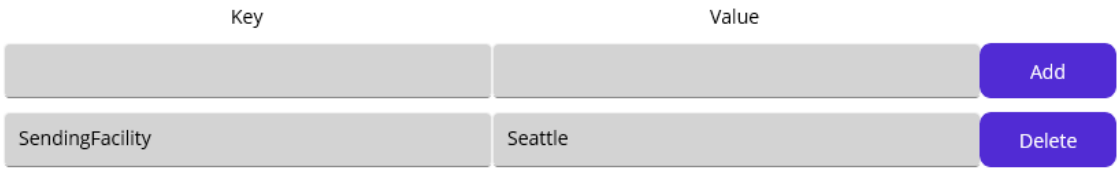


Figure 203.Added Instance References

As you can see, you can delete any of the added References by clicking the Delete button on the right. Or, you can directly modify added key and value then click the project Save ( ) button. In the same way, you can add any key that may belong to the Instance and let the Service references them. But for now, just keep this SendingFacility only.

Note: Although we don't have any Reference adding section in the Service, we call the References "Service References" because they are used in the Services only.

Accessing References from the Service

As you may already notice, all group models - Domain, Instance, Project, and Group - have a Reference-adding section. So, you can assume that each group model has its own References that can be accessed from the services. To support that, Maca provides a dedicated variable for each group model, as shown in Table 30.

Models	Accessing variables	Usage examples in the Service
Domain	_domain	_domain["Key"]
Instance	_inst	_inst["SendingFacility"]
Project	_proj	_proj["ProjKey"]
Group	_group	_group["TargetService"]

Table 29.Group Model References

In Figure 203, we assigned SendingFacility to the Instance. So, we can use that reference, like _inst["SendingFacility"], in the DICOM Reader service, as shown in Figure 204.

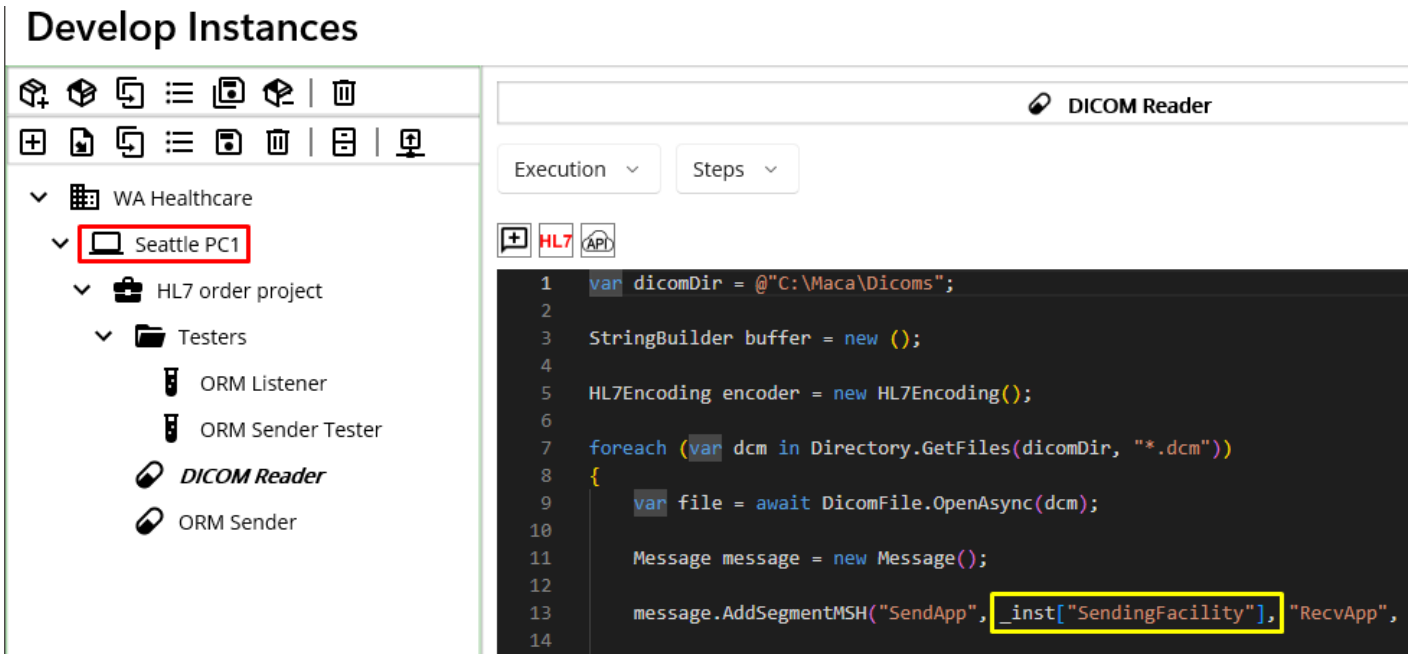


Figure 204.Added Instance Reference

We can apply the same reference to the ORM Sender Tester, as shown in Figure 205.

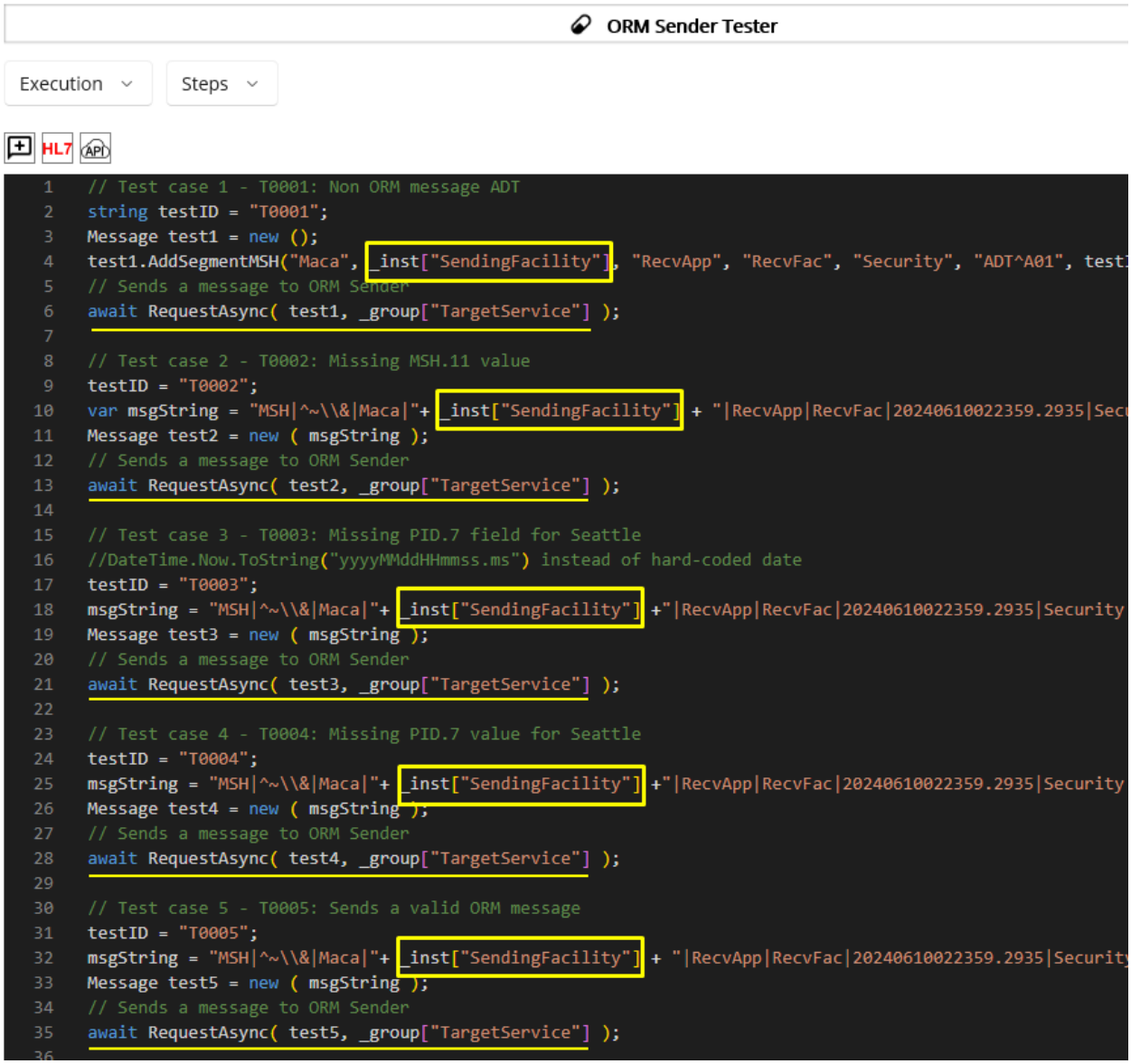


Figure 205.References Added Service

Compared to Figure 174, you will notice that each hard-coded value “Seattle” was replaced with _inst[“SendingFacility”]. In addition, each Request Mapping code was replaced with a group Reference, _group[“TargetService”], as shown in Figure 206.

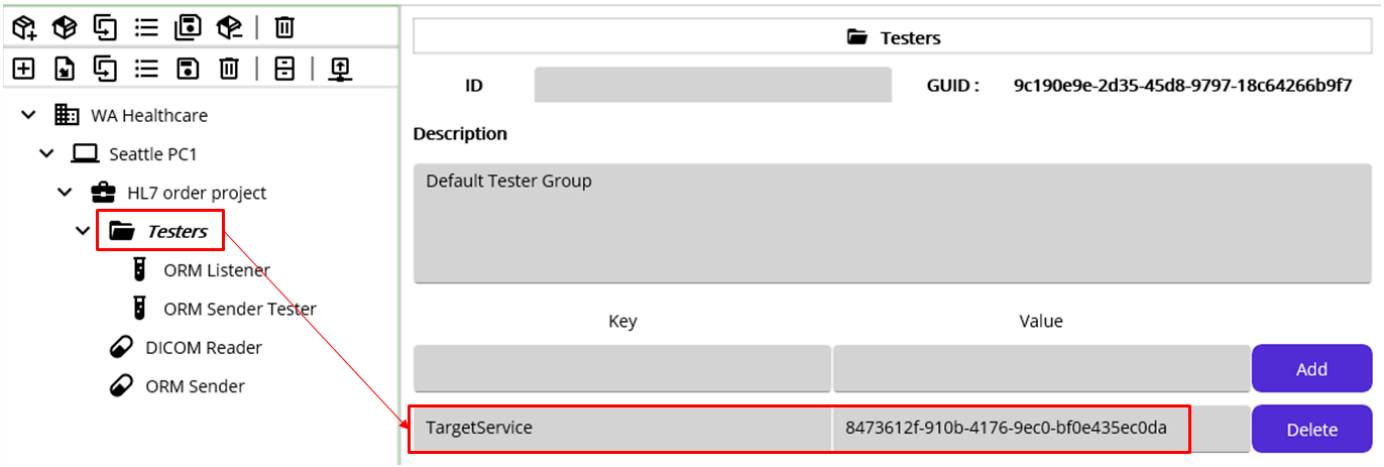


Figure 206.Added Group Reference

If you look at the Figure 174 or below Figure 207, you will notice that the value is the GUID assigned to the ORM Sender.

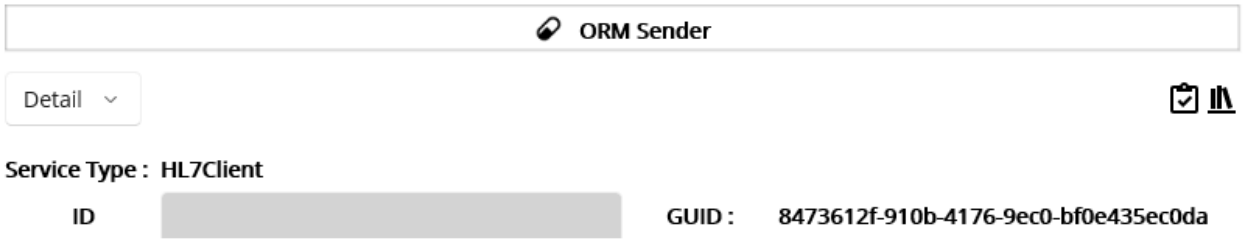


Figure 207.ORM Sender's GUID

So, you will notice that _group[“TargetService”] gets the GUID from the Testers’ Reference and set in the 2nd argument of the Request method. As we set the ORM Sender Tester pointing at the Instance Reference and the Group Reference, we now have an ability to modify the value without modifying the ORM Sender Tester when the values are changed.

Reference Hierarchy

Although the service can access the References, it is possible only when they are in the same pipeline.

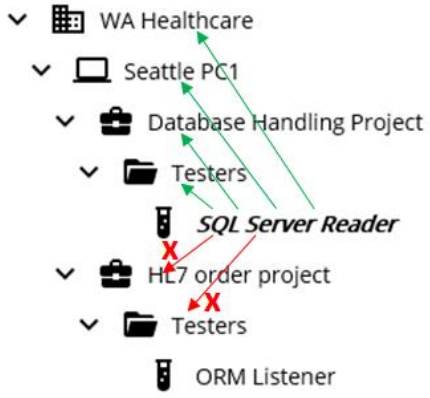


Figure 208.Accessible and Inaccessible References

If we have two projects like Figure 208, each project’s services can’t access other project’s group References. For example, the SQL Server Reader service can’t access HL7 order project’s and Testers’ References because they are in the other pipeline. In other words, the variable defined in Table 30 goes upward only. So, by simply using reference variables, you can easily get the right group model’s values without worrying about the hierarchy. As long as you put the relevant key and value set in each group model, Service References will be very useful in terms of service maintenance.

Update a project: Scenario 2

Directions

While you developing initial assignment, the client slightly changed the requirements. They changed the ACK to send a custom message based on the results. However, the specific results are not defined. So, you have to test the service with your own test cases to handle custom ACKs. The second one is that they set the location values in their system and want the value as their internal surrogate id instead of string value in MSH.4. For example, 1 for Seattle and 2 for Bellevue.

Objectives

For simplicity, your application will do the following;

- Set the MSH.4 to 1, which is Seattle’s assigned ID, and update ORM Listener accordingly.
- Test previous cases to support custom ACK messages.

Update Services to populate 1 in MSH.4.

As we used Service References for the DICOM Reader, we can modify the value in the Instance model to support surrogate ID for Seattle. Open Development workstation. From the project explorer, select Seattle PC1 node. As the Reference can be directly modified, update the value Seattle to 1, as shown in Figure 209.

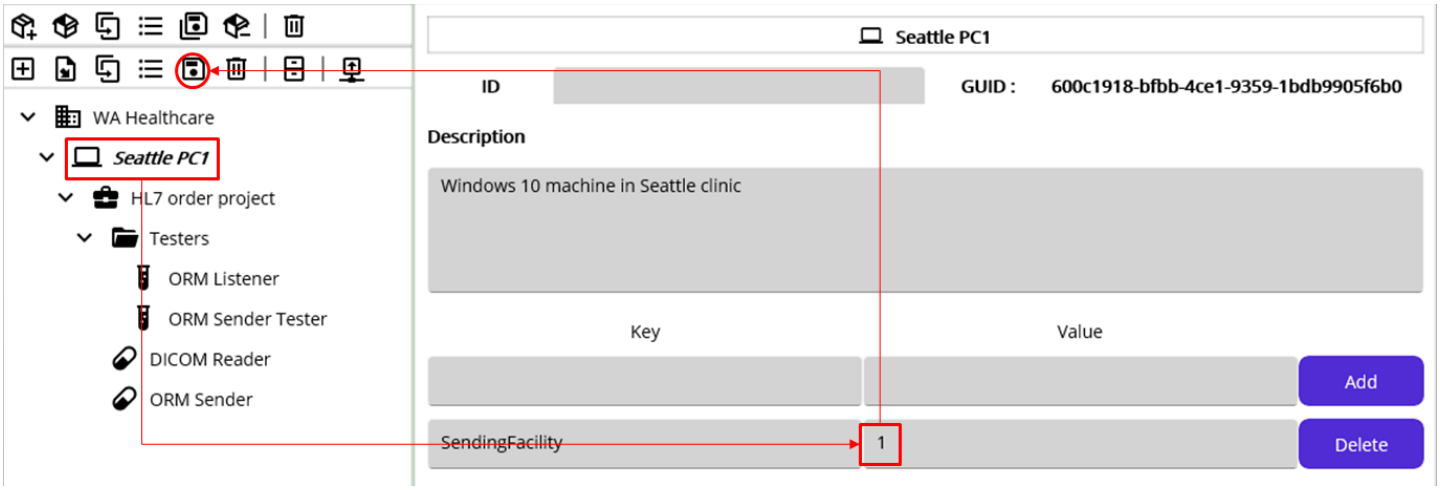


Figure 209.Modifying Instance References

As _inst["SendingFacility"] is in the DICOM Reader, this change will affect the service as well. So we don't have to change anything in the service. Likewise, we replaced all MSH.4 value with the Reference in the ORM Sender Tester, the change will be affected automatically, i.e., MSH.4 will be populated with 1 instead of Seattle.

Note: For this tutorial purpose, we intentionally set MSH.4 to be changed while you implement the services. In a real-world scenario, it will not be easy to collect all possible key values to store in the References to prepare such cases. However, if you start adding the values, it will give you an idea of what might be the most common target key terms to be kept in the References.

Next thing to do is testing existing test cases with custom ACK option. Since we don't have specific test requirement from the client, we can reuse previously used test scenarios.

Modify Service Tester

As the client wanted to receive 1 for Seattle and 2 for Bellevue, we need to change the service tester, ORM Listener to support that value.

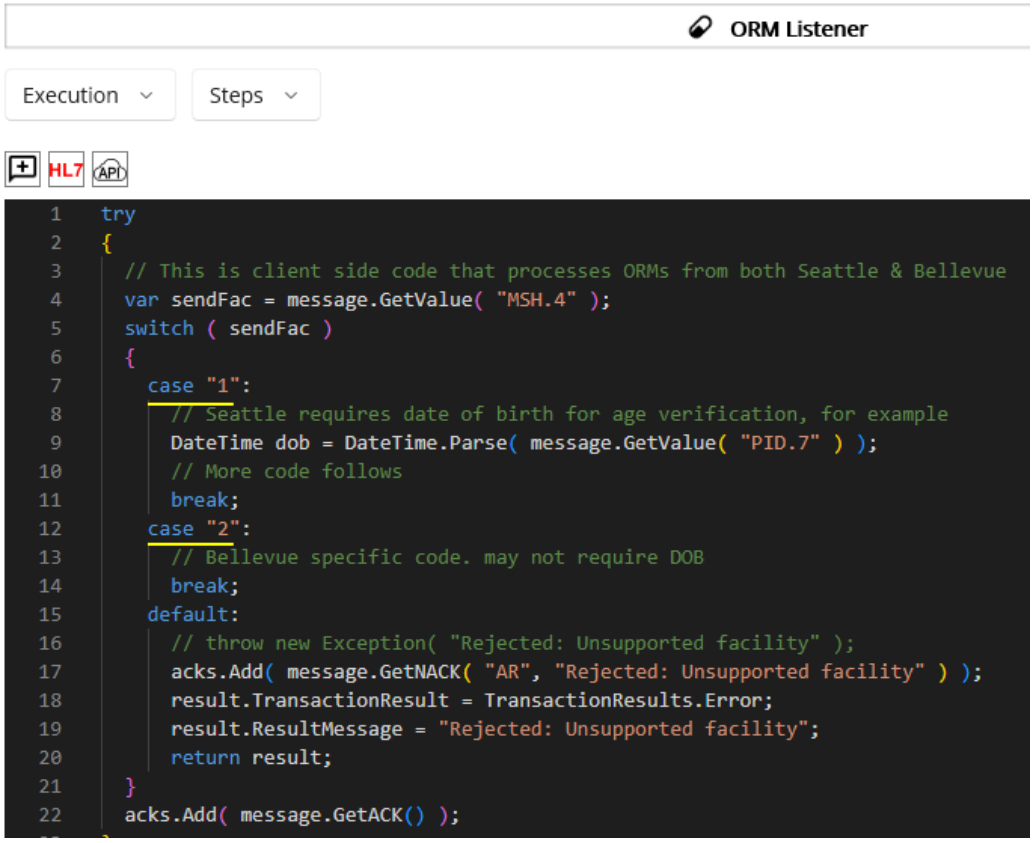


Figure 210.Updated cases for Sending Facility

Since this service simulates the client applications, we may assume that the receiving application checks the value with assigned surrogate ID instead of location name. So, we can change the values with numeric string, as shown in Figure 210. No other change is necessary for testing purposes.

Regression Test

As the client added a new requirement that is related to the existing process, we need to test previously conducted test cases to see if any issue happens when we redo the test. Although we need to check MSH.4 and custom ACK, the main logic that we implemented in both required service and tester service will be almost the same. Of course, we didn't add further code that needs to be done based on the ACK results and thus the test is incomplete in fact. However, this tutorial is not targeted to go over complete test methodologies and rather to use Maca application as a simple test tool. So, we recommend you extend Maca with your own idea to test the service thoroughly once you finish this tutorial.

Since we set the Default ACK switch on, let's switch that off first. Select the ORM Listener node from the project explorer. Then go to Settings > Config view using Service Navigator. Then set Send switch off, as shown in Figure 211.

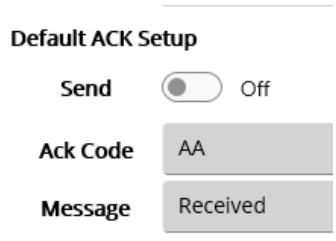


Figure 211.Send Default ACK switch off

Once you set this, click either the Save ( ) button or Verify ( ) button to make sure the change is in effect. Then click the Load ( ) button. In addition, we didn't load the models where the References are applied. So, please load Seattle PC1, Testers, DICOM Reader, and ORM Listener services just in case. Once you load all related models, switch to the Management workstation.

Deploy updated models

Click the Source tab if the Instance tab is currently showing. To make sure the server shows the latest image after the model load, click the Refresh ( ) button. As we went over the issue that happens when deploying the domain model, we can deploy the service one by one at this time. First of all, we need to load the Instance model without the children. Select the Instance node and click the Deploy ( ) button. When the popup shows up, click the No button, as shown in Figure 212 to deploy Instance only.

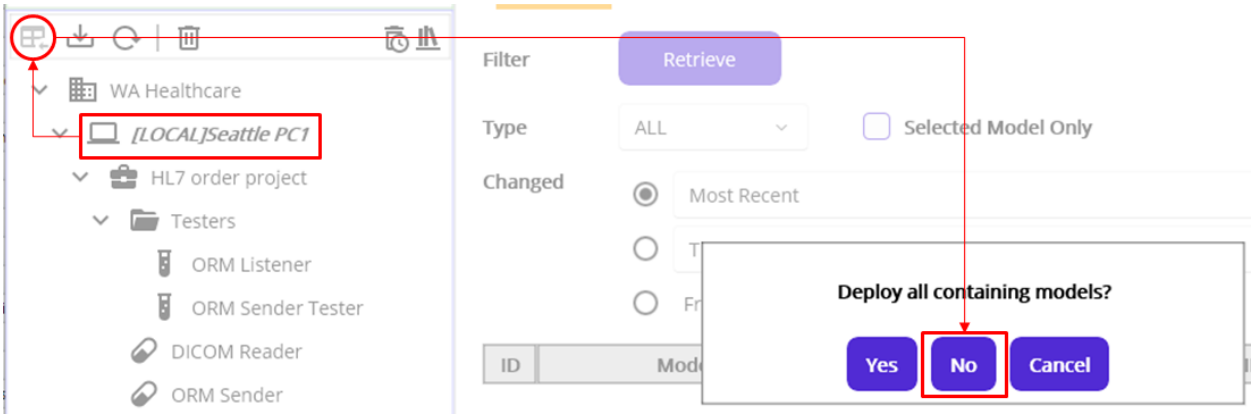


Figure 212.Deploy Instance model only

Based on the dependency, select the ORM Listener node and click the Deploy ( ) button. Then deploy the ORM Sender Service. After that, we need to check whether the updated DICOM Reader works as expected. So, deploy the DICOM Reader lastly.

Verify the DICOM Reader data

Once the deployment is done, click the Instance tab. Since the DICOM Reader will execute the code when deployed by design, please stop the service first to prevent the test data from piling up because the service keeps repeating unless you remove the DICOM file from the destination folder. Then select the Most Recent picker and click the Retrieve button, as shown in Figure 213.

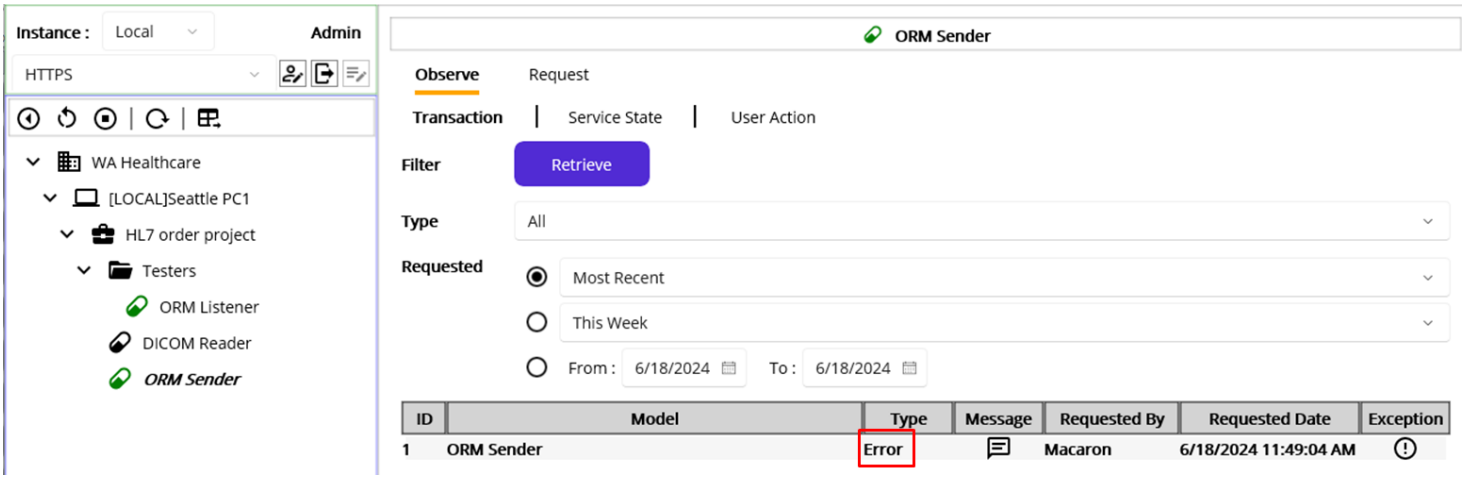


Figure 213.ORM Sender transaction using DICOM Reader's ORM

As you may know, there is no date of birth adding code in the DICOM Reader service. And the ORM Listener requires the value. So, the transaction should be marked as Error as expected. You can click the Message and Exception button to see the details, as shown in Figure 214.

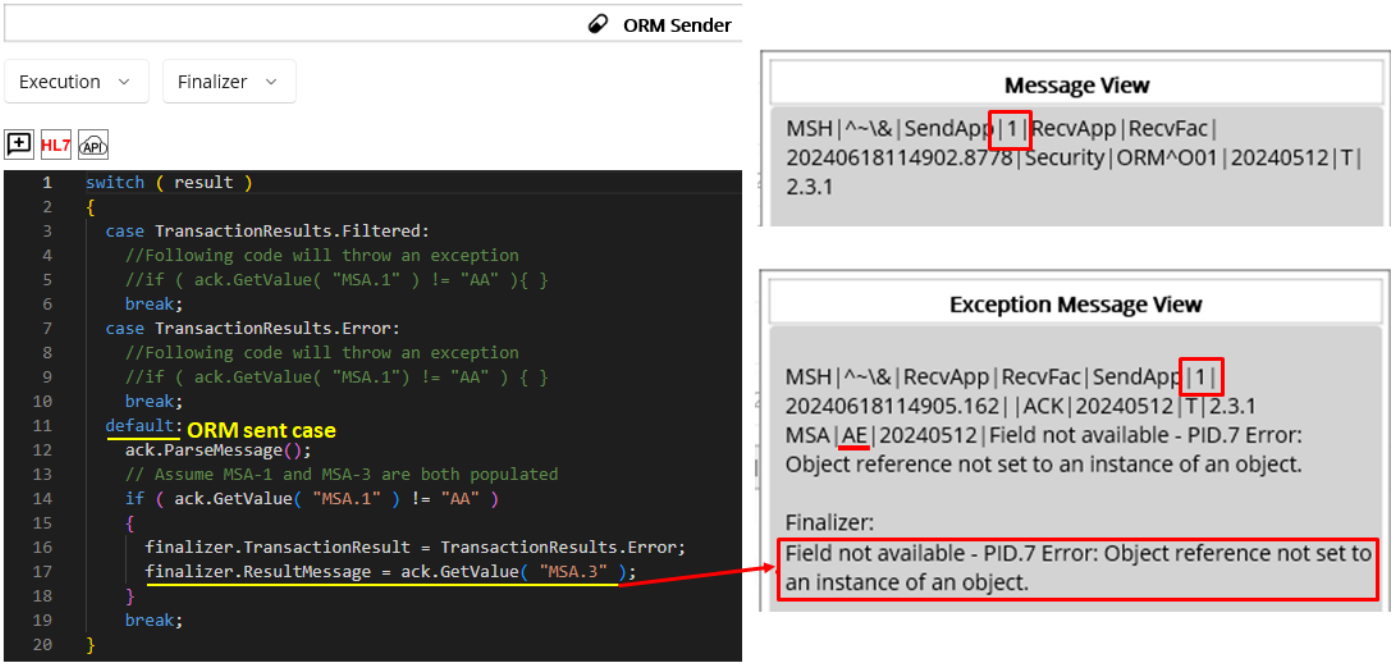


Figure 214.ORM Sender's Finalizer with Message and Exception

In the Finalizer, we put the default case that reaches when ORM was sent. And if the result is not AA, then the code adds received ACK's MSA.3 to the ResultMessage that can be saved as the last part of the Exception message. As you see in Figure 214, the last line below Finalizer in the Exception Message View could be redundant because the above ACK contains that message as well. But this is for demo purposes to show how to use the Finalizer, we can keep the code as is.

Since we didn't add the PID.7 in the ORM, the error happened here is exactly the same as the 3rd test case in the ORM Sender Tester. If you see Figure 170, you will notice that the HL7Exception filter handled the issue occurred when the code tried to get missing PID.7 field's value. To cross check the record, select the ORM Listener and click the Retrieve button to retrieve the Most Recent record, as shown in Figure 215.

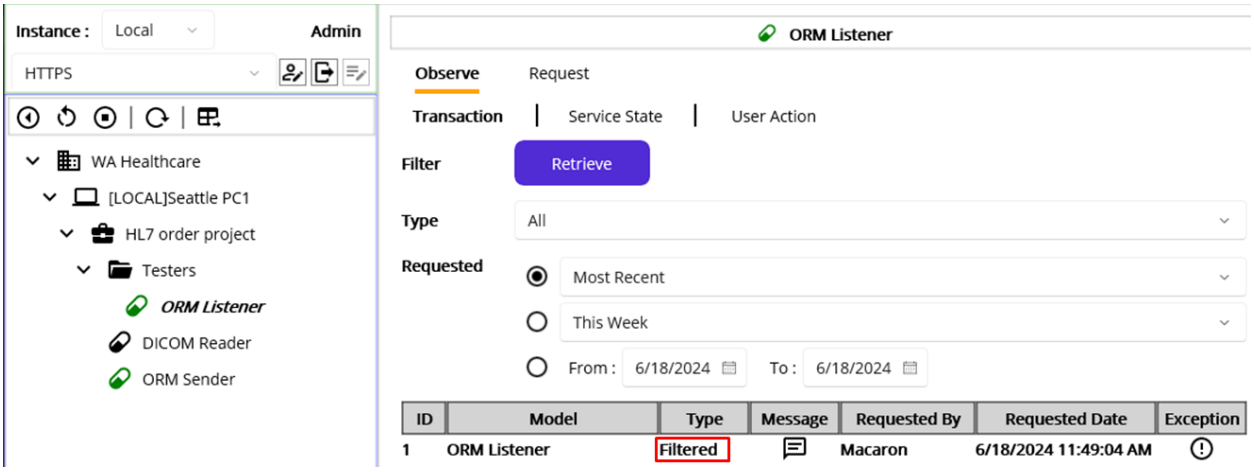


Figure 215.The most recent record in the ORM Listener

As you see the record's Type is Filtered, we can click the Message and Exception buttons to see the detailinformation, as shown in Figure 216.

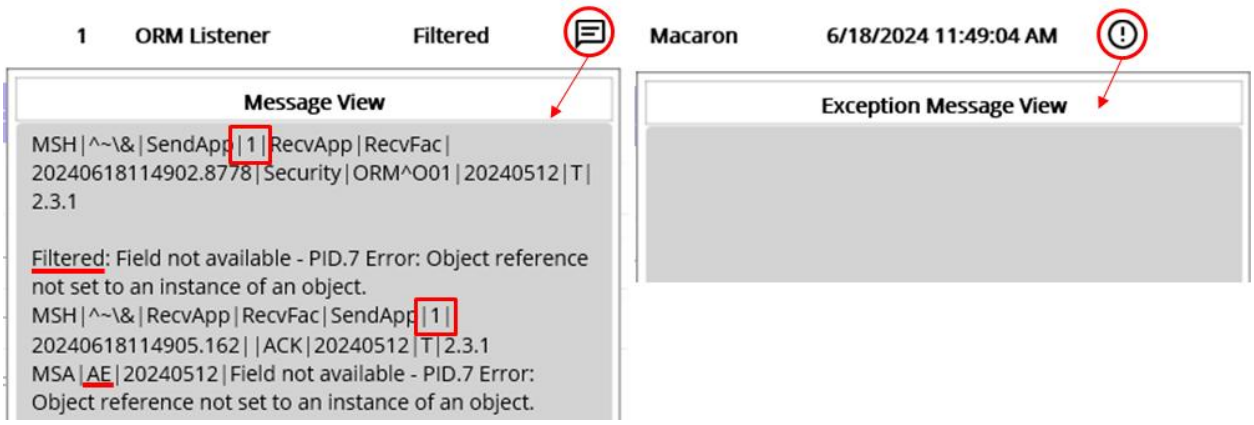


Figure 216.Filtered transaction's Message and Exception

If you compare the Message here with the Exception in Figure 214, you will know that the ORM Listener processed the message as designed. We could see the sending facility is successfully populated with 1 and missing PID.7 made the message filtered in the ORM Listener. And the ACK was returned with AE and HL7Exception's Message and the ORM Sender marked the transaction as Error in the Finalizer correctly. Although we didn't document this to track issue later, we could verify what was happening with the DICOM Reader's ORM message.

Rerun Test Set

Since the ORM Sender Tester contains 5 test cases with pre-defined ORM messages, we can rerun the service to do a regression test that will show what would be the result after the minor changes along with custom ACK. We can revisit Table 26 for this test case scenario. As we expect, there should be 3 ACK messages; 2 NACK and 1 ACK. As we already deployed the ORM Listener and ORM Sender, deploy the ORM Sender Tester now. Click the Source tab. Then select the ORM Sender Tester service and click the Deploy button. Like the DICOM Reader service, the ORM Sender will execute the code immediately when it is deployed. So, once the deployment was done, click the Instance tab. Even if the interval of the ORM Sender Tester was 30 minutes, stop the service just in case. Then select the ORM Sender and click the Retrieve button to retrieve the Today's records, as shown in Figure 217.

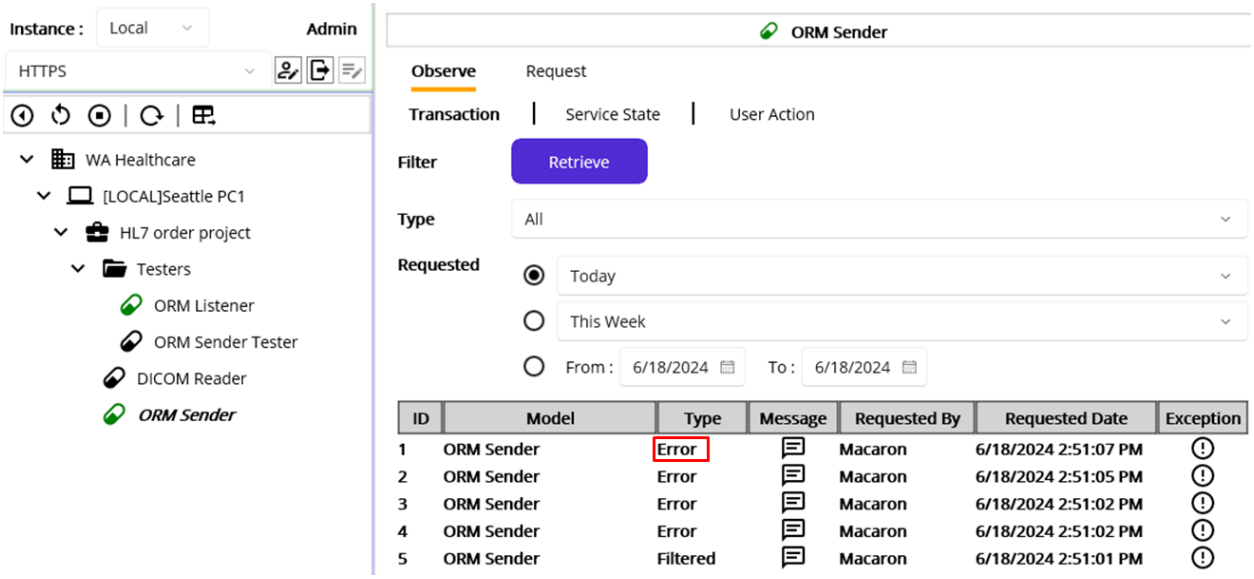


Figure 217.ORM Sender's results for ORM Sender Tester

As you may notice, among the records, the 1st record, which was the 5th test case that supposed to be Succeed, was marked as an Error. Unlike the typo issue that happened in Figure 190, we changed the string value to a numeric value, like 1, using Instance Reference. So there should be no typo issue in the ORM Listener, but somehow we encountered the Error with that message. To troubleshoot, let's click both Message and Exception buttons, as shown in Figure 218.

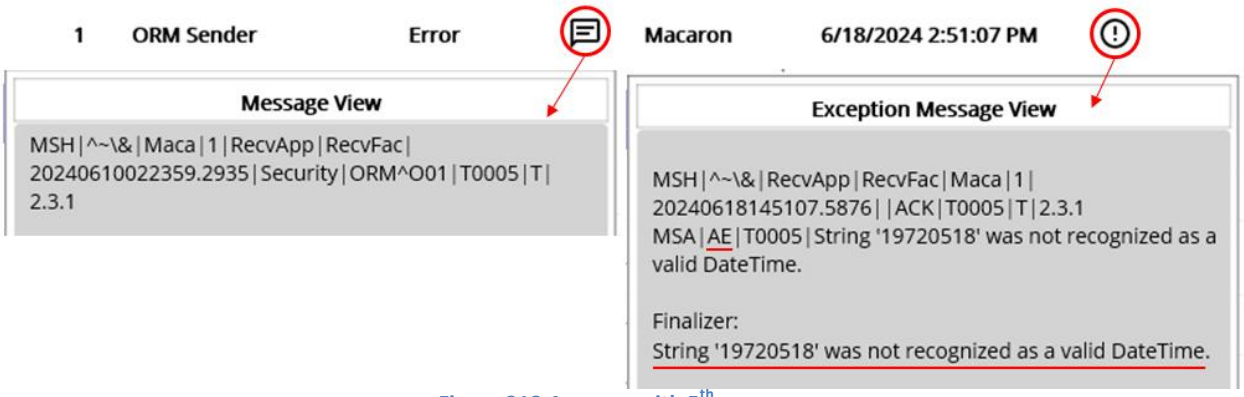


Figure 218.An error with 5th message

If you look at the Exception View that contains the received ACK's MSA.3, you will see the date of birth caused the issue. Even if the date value in PID.7 looks valid with the format of "yyyyMMdd," we received the invalid DateTime format error from the ORM Listener. While you handle the date value in the HL7 message, you will see that the date format is "yyyyMMdd…" unlike "MM/dd/yyyy" for example. Although the DateTime.Parse() supports "yyyyMMdd" format, we somehow encountered that error anyway. Once you research the issue, you will find the reason that is related to the system's culture information. Since you will handle various devices, this could happen someday. So, you may take note of this to troubleshoot the issue later.

In order to troubleshoot this, **switch back to the Development workstation**. Then select the ORM Listener and go to Execution > Steps view using the service navigator.

If you look at Figure 170, you will find the commented line below the DateTime.Parse method. Please uncomment the line and comment, or delete, the above line, which is Parse method, as shown in Figure 219.

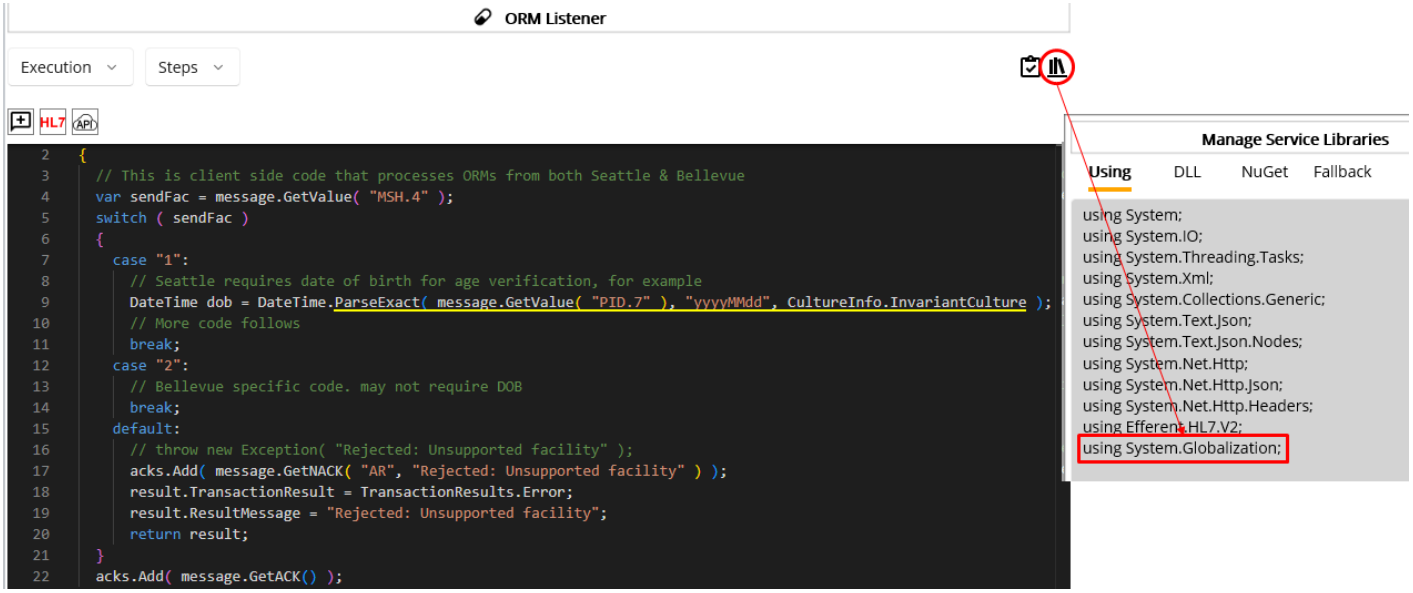


Figure 219.Use ParseExact and Globalization namespace to handle HL7 date value

One of the solutions to handle HL7 date value is using ParseExact method. In that case, we need to add System.Globalization namespace in the Using tab of the Manage Service Library popup, as shown in Figure 219. Modify the ORM Listener accordingly, and verify the service. Then load the ORM Listener. Once the service loaded, **switch back again to the Management workstation**.

Click the Source tab if you're in the Instance tab. Click the Refresh button to show latest services. Then select the ORM Listener service and click the Deploy button. Once the deployment is finished, click the Instance tab.

From the model explorer, select the ORM Sender Service that's currently stopped. Then click the Start ( ) button. That will execute 5 test messages. To make sure the tester not interfere other services, stop the ORM Sender Tester. Then select the ORM Sender and click the Retrieve button to retrieve Today's result, as shown in Figure 220.

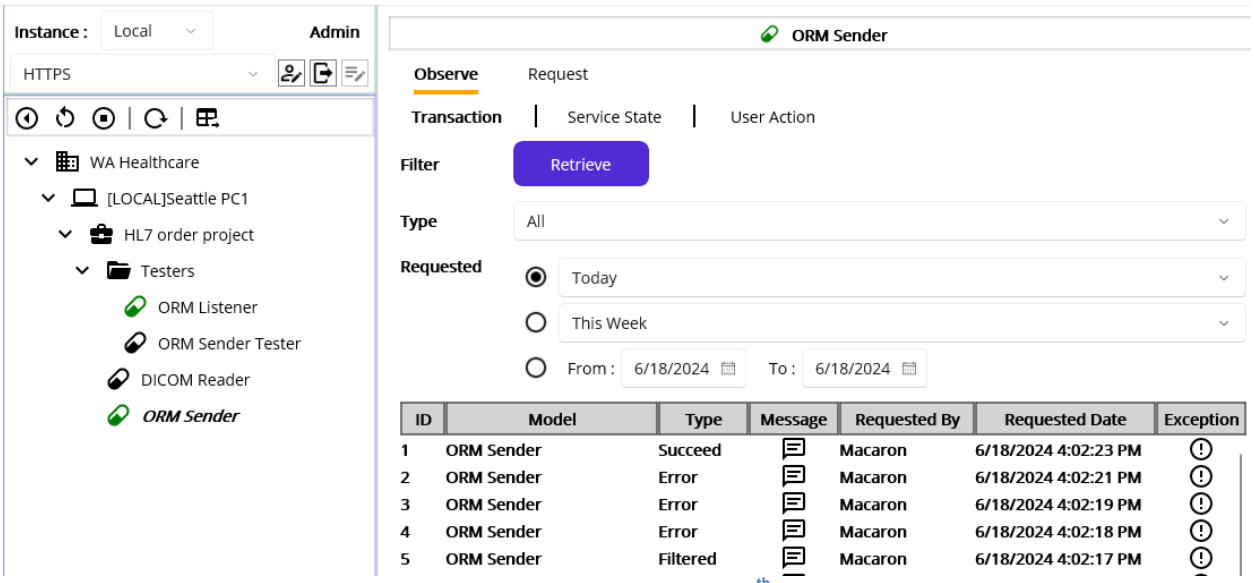


Figure 220.Troubleshooted date value in the 5th test case, i.e. ID 1.

Now, let's click the 1st record's Message and Exception buttons to see the differences, as shown in Figure 221.

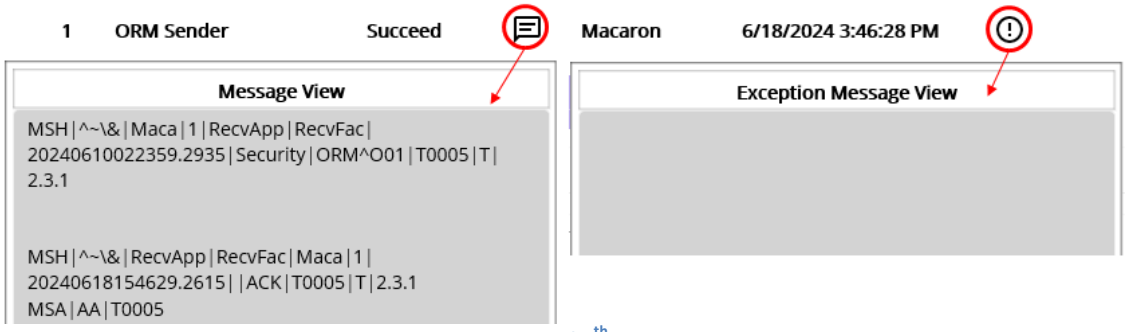


Figure 221.Succeed 5th test message

As we expected, 5th test message processed successfully in the ORM Listener and returned AA type ACK as a result. In addition, what we modified for sending application worked well by populating 1 in the ORM's MSH.4 and the ACK's MSH.6.

Now move on to the error cases and check the differences by retrieving the ORM Listener, as shown in Figure 222.

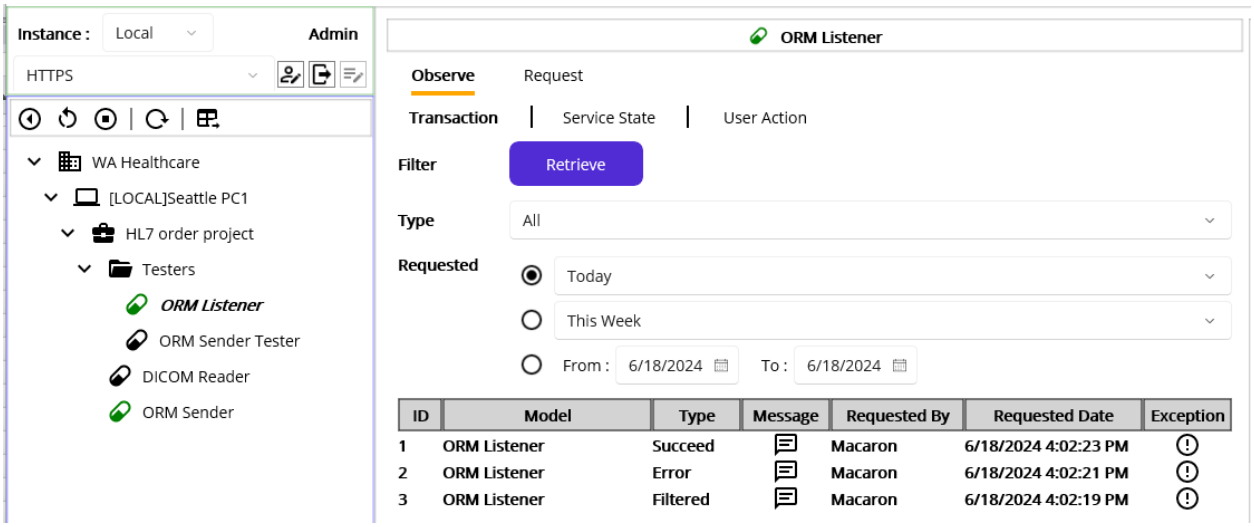


Figure 222.ORM Listener results with 3 test messages

As we know, the first two test messages were filtered in the ORM Sender and the rest 3 were sent. But if you look at the 3rd record in the ORM Listener, you will notice that the message was filtered. Click the Message and Exception buttons to see the detail.

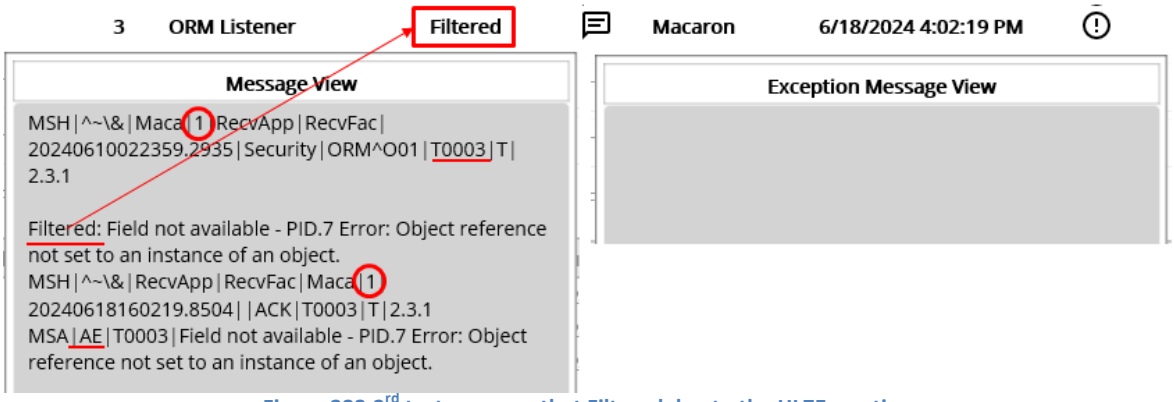


Figure 223.3rd test message that Filtered due to the HL7Exception

As you see in Figure 223, HL7Exception handled missing PID.7 error and set the transaction as Filtered, which is as expected. But in the ORM Sender, the Finalizer's default case checks the returned ACK's MSA.1 value. And that is not AA, set the transaction as Error, as shown in Figure 224.

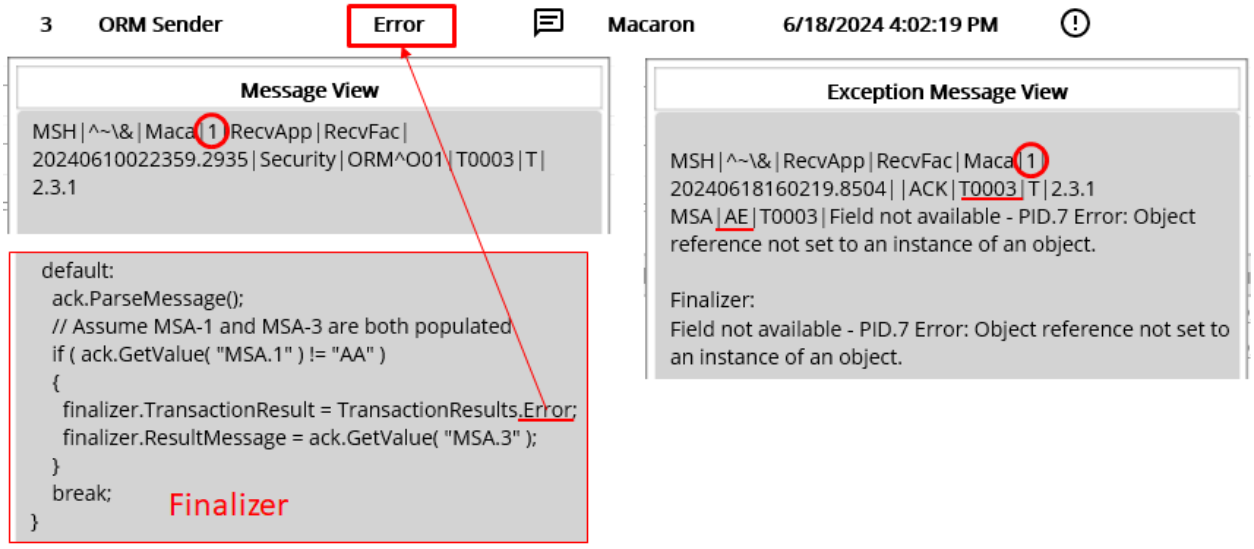


Figure 224.Sets the transaction as Error if NACK is received

Although both ends show different types, it is as designed and we can consider this as passed. And the sending application value 1 is populated well in both ORM and ACK messages. We are not testing the ORM Listener, but cross-checking both ends will give more robust test result.

Now, let's move on to the last message which is the 2nd result in the ORM Listener.

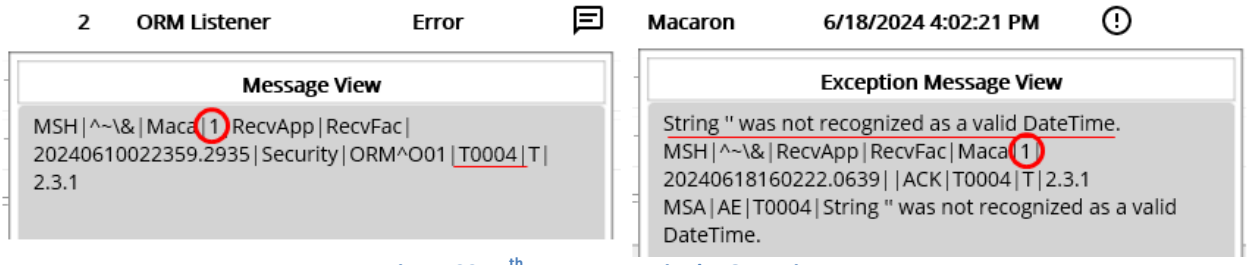


Figure 225.4th test message in the ORM Listener

Although we updated the ORM Listener to use the ParseExact, still we pass the empty string and thus Exception was thrown. Then the transaction marked as Error and NACK was returned as expected.

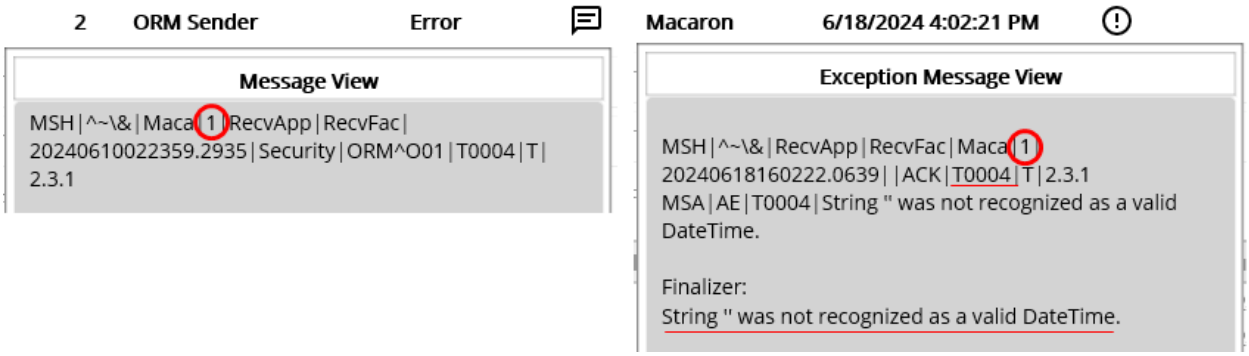


Figure 226.4th test message in the ORM Sender

Similar to the previous case, sending facility value has processed successfully. And the error was handled in the Finalizer successfully by marking the transaction as an Error. And the error message was produced in the Finalizer as expected, as shown in Figure 226.

Service development approaches

We just finished the very basic testing and the result verification using required and tester services. Based on this, you can extend what was shown here or implement new services that meet your requirement. As an extra guideline about the test service, you may think of following same service implementations.

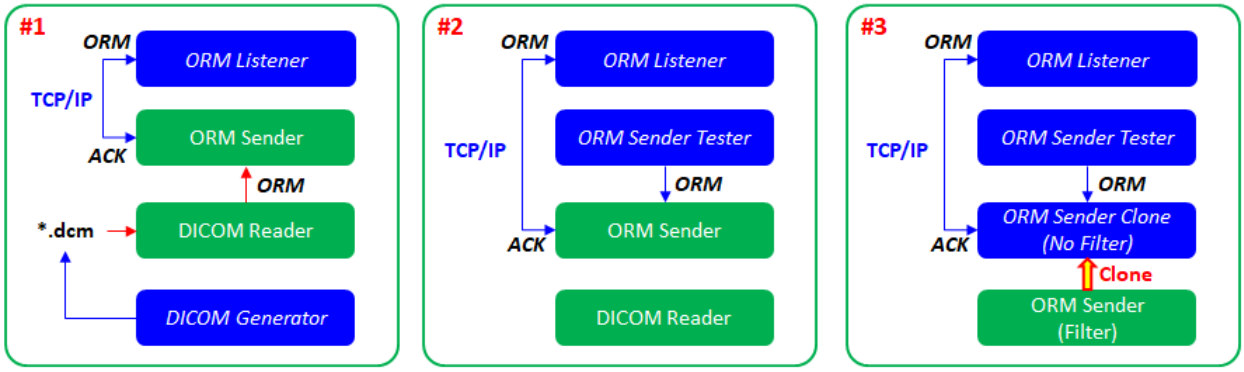


Figure 227.Service implementation examples

#1 was what we didn't take into account when testing the ORM messages. This approach will give you extra practice in handling DICOM files by using fo-dicom that was introduced earlier. The dll provides helpful method to modify the DICOM files Tags, and thus you can modify and save the DICOM for your practice or testing. As the blue color means the tester service, you need to put that

service under the Testers folder to do not interfere other required services. Then you can reuse the ORM Listener that we handled in this tutorial.

#2 was what we implemented in this tutorial. We still can use DICOM Reader service, but the main service will be ORM Sender Tester service. You can revisit above two scenarios for this approach.

#3 is a little bit test- or debug-oriented approach. In case you need to check the finished service and want to modify heavily to check or verify something, you can clone the required service and put that in the Testers folder. For example, if you want to test specific case but don't want to go through entire code, you can clone and comment or delete unwanted code and let the service run. By doing that, you can make the required service intact. Although you can use external source management service, such as git, to make the original service intact, you can take this approach that doesn't affect other required services because these services are not auto-deployed even they are loaded on the server.

Clone Services

If you want to clone the required service, i.e., the ORM Sender, select the service from the project explorer first. But before that, you need to be on the Development workstation first. Then click the Clone Selected Model in the project toolbar, as shown in Figure 228

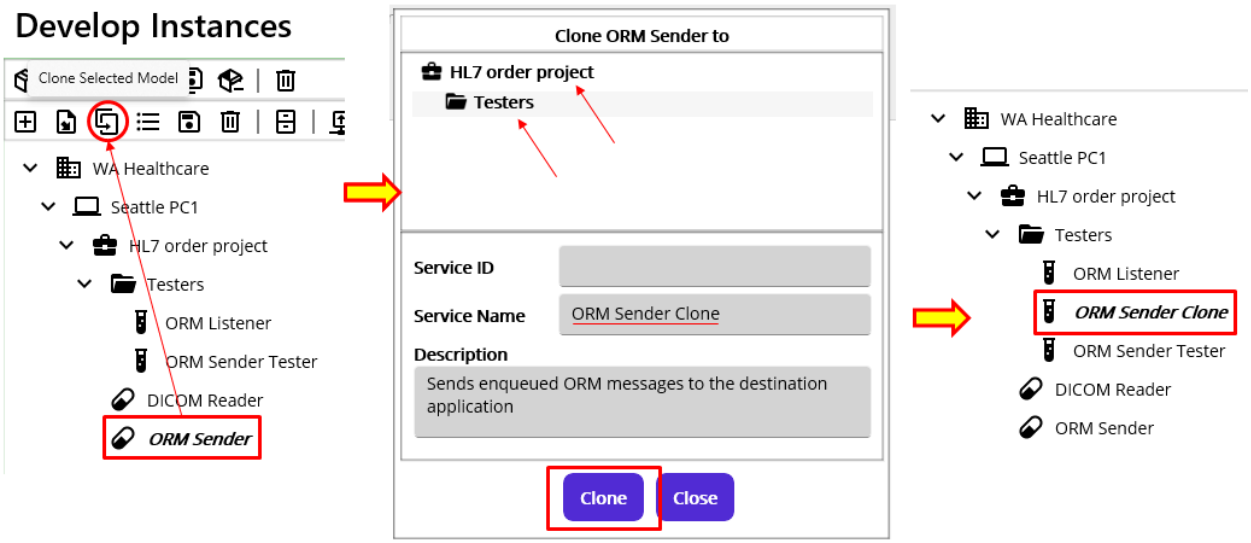


Figure 228.Clone Macaron Service

Once you open the Clone service popup, you can select possible target location to clone. In this case, the Testers folder was selected and the Clone button was clicked.

Clone from the recently created or cloned model

If you don't have a model in the project, you still can clone a model from previously created or cloned model. As we did, ORM Sender Clone service was cloned, and thus the record was saved in the recent models, as shown in Figure 229. What you need to do first is to select the target folder that you want to clone. And the target locations are Project or Group. In this case, we selected HL7 order project. So, when you click Clone button, another clone file from the previous history will be cloned to the selected location.

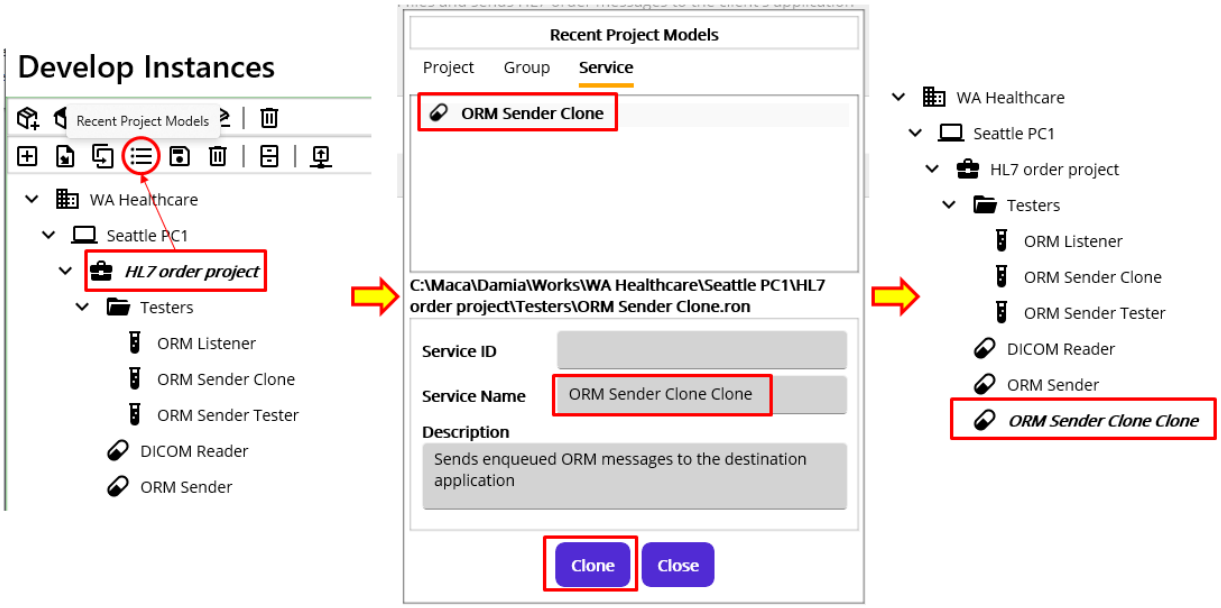


Figure 229.Recent Project Models

For the clarification, we set the name little bit weirdly, by appending Clone twice. By using this, not only for the testing purpose but also for the reproducing a model that shares the same code could be easily done in the specific target location.

Likewise, if you created the Project or Group, you can select one of them from Recent Project Models popup and close that model.

Import Project Model

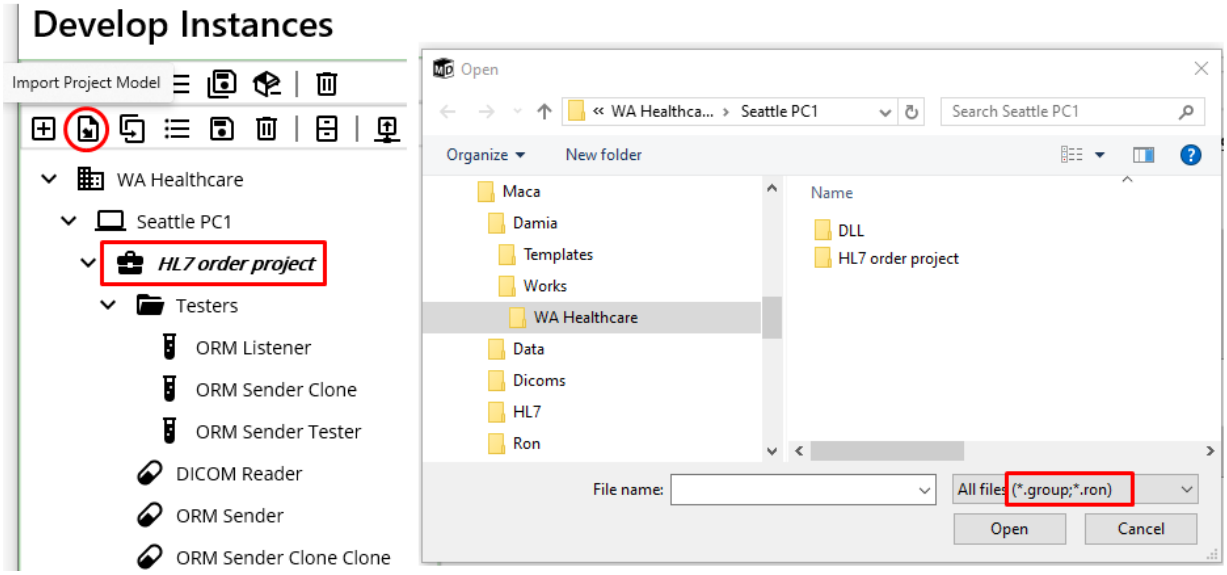


Figure 230.Group or Service mode import under the project

If you don't have a model in the loaded instance or in the recent list, you can still import the project model. Based on the selected model on the project explorer on the left, the popup shows only available model list to import. For example, since the selected model is a project, only service group or service could be selected, as shown in Figure 230. Likewise, if you select a service group, only the service could be selected, as shown in Figure 231.

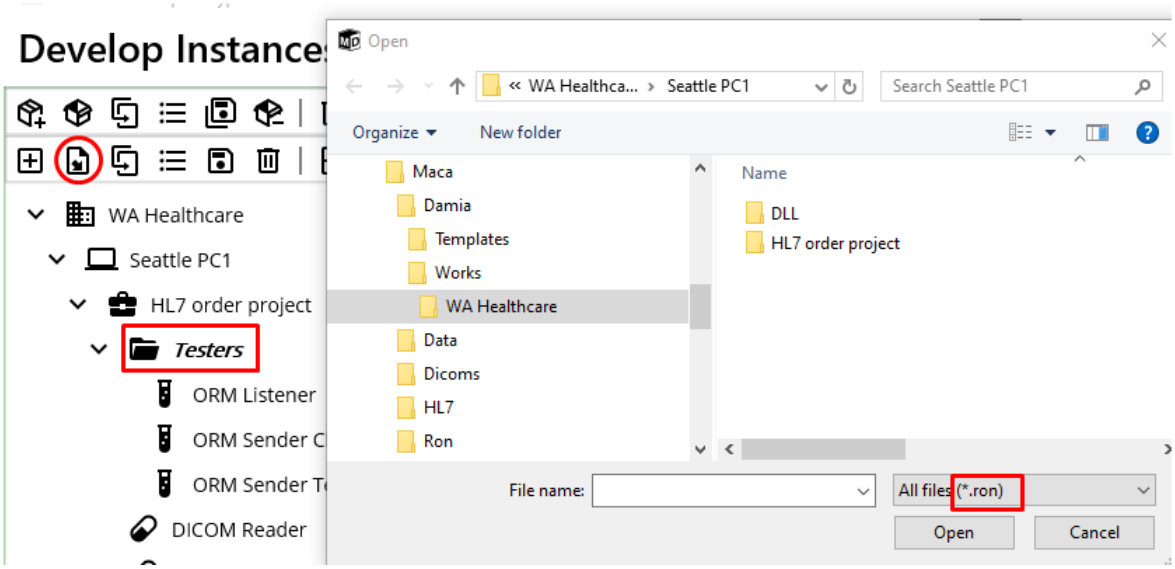


Figure 231.Import a service under the Service Group

Since this is importing project model, if you select the Domain or Instance, only the project can be selected.

Save frequently used model to the Templates

If you have a model that could be used frequently, you can register that model to the Templates and add it later. For example, if the HL7 order project is common project that can be used in multiple projects, you can select the project and register, as shown in Figure 232.

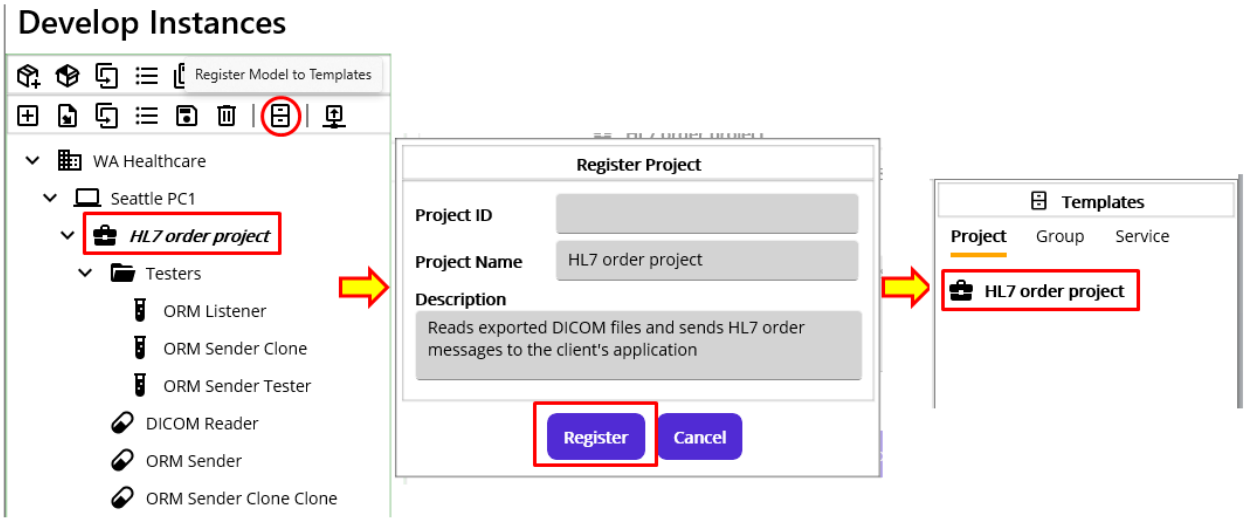


Figure 232.Register a project to Templates

Once you have a template, you can add selected template to the loaded Instance on the left, as shown in Figure 233. But please note that the selected model on the project explorer is restricted based on the selected target model type. For example, if you select group model from the project explorer and try to add project model, it will not be possible. Only when you select higher level model, then you can add selected template model from the Templates.

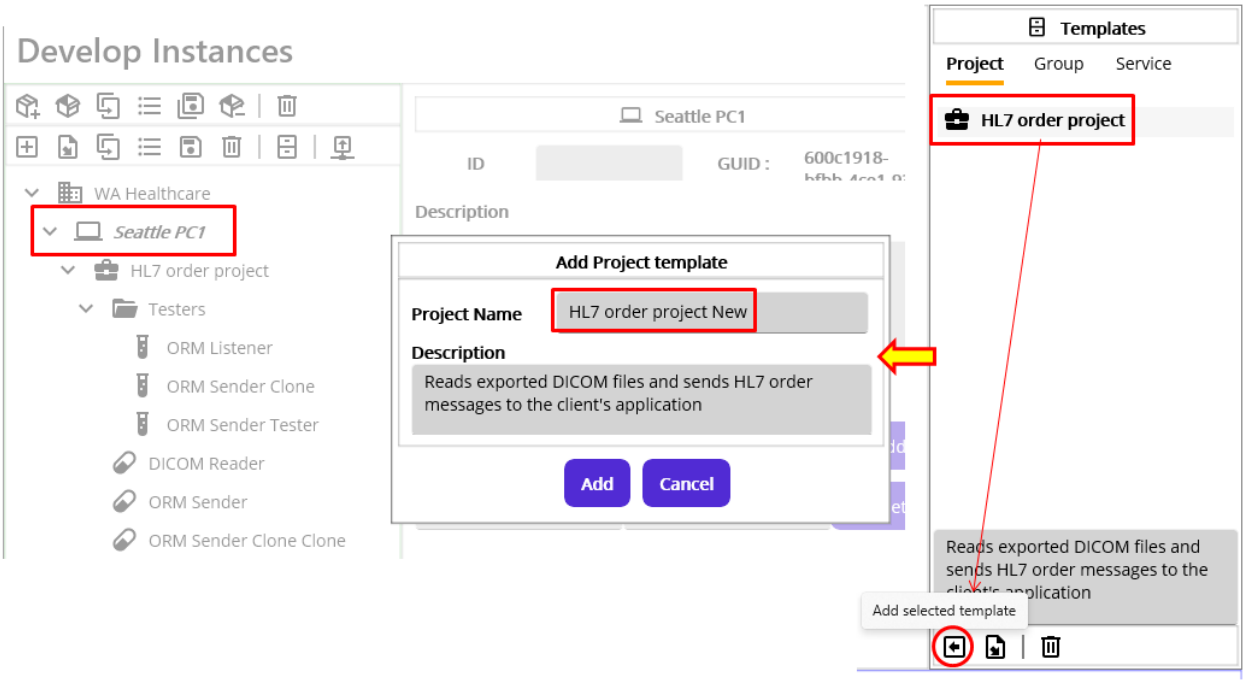


Figure 233.Add a Project Template to the Instance

Similarly, you can add a Service to the Templates as well. But at this time, you can register the service under the group, as shown in Figure 234.

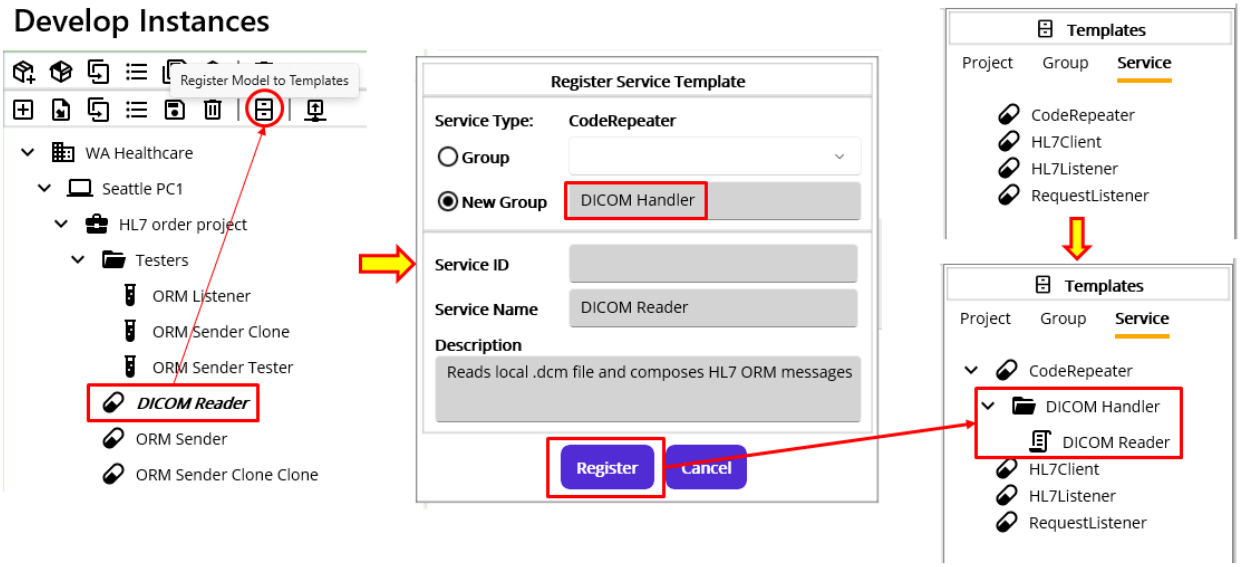


Figure 234.Register a Service

In the same way, you can register the other service types, such as RequestListener. By using this structure, you can organize the service well.

Additive messages

So far, we have verified the results of the updated project according to the client requirements. In the meantime, we fixed the additional errors that caused by some methods and did the simple test and regression test to verify the result again. Although we didn't finish the objectives completely, the steps we followed so far will give you how to use Maca and what might be considered while using it. In addition, some introductive guideline that you may take advantage of will also give you an idea how to extend Maca for your future projects. Since this is an initial prototype that didn't have entire features that we planned, you may find the upgradeable or additive features of Maca, not to mention the bugs or errors. There is a multiple HL7 message handling feature in

Maca, but it is not fully implemented yet. So the next release will cover that along with HL7 batch message. So, please feel free to let us know any of them.

Update a project: Scenario 3

Directions

WA Healthcare will launch a new cloud service that handles the patient-related files. The Seattle client wants to use that service to upload the patient's images and download the image-related PDF files from it. The cloud service will provide those features as a **Web API**[36]. Since the service is not in service yet, the destination information, such as address and service names, is not provided.

Note.The content of the PDF is not specified here as it is a hypothetical project. As the information related to the image could be seen and downloaded on the website or app, it may not be necessary to download the same information as a PDF in a real scenario.

Objectives

For simplicity, you are assigned to implement the following new features:

- Upload the patient's image extracted from the DICOM file to the cloud.
- Download the PDF files from the cloud to the current machine.

Web API and HttpClient

As we are going to use Web API services, it's highly recommended to check out the client-side Web API consumption. And there is a good tutorial site to start with: https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/http/httpclient. If you are not familiar with the **HttpClient**[37] that Maca heavily uses, you can spend some time looking through it. As this tutorial uses many parts of the site to demonstrate Damia's features along with the sample projects, it will be a good idea to keep checking that site for further references. If you are still unfamiliar with its use, the following sections will guide you step-by-step through examples on how to use HttpClient in Maca. So, let's start building the services!

Create a sample project for web API

Before we implement the actual requirements, let's create a sample project to practice how to use HttpClient in Damia. As you saw the HttpClient usage examples on the tutorial site, we will reuse many of them to make you feel more comfortable with it. In addition, we are not going to create any extra web API project to test with our services because we keep reusing the jsonplaceholder site (https://jsonplaceholder.typicode.com) as our target destination. So, we can focus on the services only.

Create a new Project

Instead of putting new services under the existing project or group, we may place them under the separate project to make them more identifiable and manageable. But before we create the project, let's think about the name of the project.

Once you start composing HTTP requests like GET or POST, you'll notice that the base address needs to be defined first. Because that contains all HTTP methods, we can convert that base address to a meaningful name for the project. So, let's create a new project that contains the target destination information, such as "**Jsonplaceholder Web API**." By doing this, we will easily know that all services under this project are web APIs belonging to the jsonplaceholder site.

Note. "**HTTP requests**" mentioned here can be interchangeable with "**HTTP methods**" or "**HTTP verbs**" throughout this tutorial.

Create a Service Group

Instead of creating a MacaronService right under the project, you can create a ServiceGroup first and then add the MacaronServices there. As you may notice that the **route** could contain the web API group information, making a ServiceGroup's name represent the web API's group information is another good way to organize your project as well as make it more recognizable. For the web API **route group**, you can check this site: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/route-

[36] https://dotnet.microsoft.com/en-us/apps/aspnet/apis
[37] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.httpclient?view=net-9.0

handlers?view=aspnetcore-9.0&source=recommendations#route-groups or https://learn.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/routing-in-aspnet-web-api. Since the route contains slash(/) to make the group clear, you can set the ServiceGroup name with your own recognizable value accordingly. For this sample project, create a ServiceGroup named "**Todos**" for simplicity. Now it is more clear that all containing HTTP methods(GET, POST, PUT, and DELETE) are for the Todos.

Note. For now, "Todos" is not quite clear what exactly it is. Once you create another group like "Users" that handles all user related requests, such as create a user or update a user, you will notice that Todos is for handling todo list, such as create a todo item or get the todo list. So, please set the ServiceGroup name based on the context.

Create a MacaronService

As we know that the destination is "https://jsonplaceholder.typicode.com" and the route is "todos" based on the tutorial site, we can use that information to compose the HTTP requests. Among them, let's start with the GET method first. In most cases, the GET method is called periodically or on the specific case based on the defined conditions. For the first sample service, make the service call the GET method periodically, e.g., every 1 minute.

Select the group if that's not selected. Create a CodeRepeater service and name it "**GET todo list**". Go to Settings > Config and set 1 minute for Execution Mode.

Naming web API consuming services

In the previous steps, we created a project, a group, and a service, as shown in Figure 235.

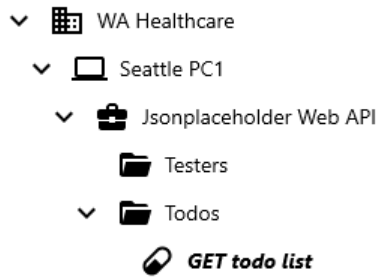


Figure 235.Add HTTP GET servcie

With this initial project structure, you may think that the other services' names may have a similar structure. Since we will try the other POST, PUT, and DELETE methods as well, we can set other services' names in the same way, as shown in Figure 236.

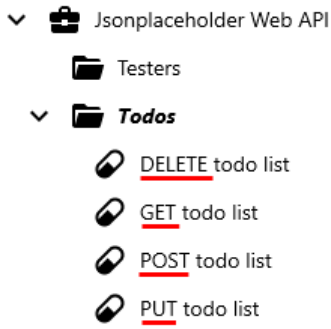


Figure 236. All HTTP method services

At first look, you can see each MacaronService starts with one of GET, POST, PUT, and DELETE. According to the above structure of the project, you will notice the following:

- The project is for the target destination, i.e., the base address.
- The group is for the route or route group.
- The service is for the HTTP method and starts with an Http verb.

In addition, the project and the group can have their own References that can be referenced in the services. We will cover this later in detail with examples.

For now, this is a very basic structure that doesn't do any other work. In addtion, it's not clear whether each service is CodeRepeater or RequestLister. So this may not meet future requirements in the end. But by keeping this structure, we can easily identify what the project, group, and internal services are for.

Execute an HTTP GET method

So far, we set the service "GET todo list" as repeating every minute. What we need to do next is to make this servcie call the GET method every minute. Select "GET todo list" service if it's not selected. Go to Execution > Steps view. In the code editor, type following sample code, which is similar to the example code int the tutorial site. You can simply copy and paste that in the code editer.

```
using HttpClient httpClient = _httpClientFactory.CreateClient();
httpClient.BaseAddress = new Uri("https://jsonplaceholder.typicode.com");
string routeQuery = "todos?userId=1&completed=false";
using HttpResponseMessage response = await httpClient.GetAsync(routeQuery);

//response.EnsureSuccessStatusCode();
var jsonString = await response.Content.ReadAsStringAsync();

string path = "C:\\Maca\\Temp";
if(!Directory.Exists(path))
   Directory.CreateDirectory(path);

File.WriteAllText(Path.Combine(path,DateTime.Now.ToString("yyyyMMddHHmmss"))+".txt", jsonString);
```

Figure 237.Initial HTTP GET code

Then click the Verify Service ( ) button to see if any compiling issue comes up. If there's no issue, you can load the project. Select the project. Then click the Load Model ( ) button from the Project toolbar. Please make sure the Macaron server is up and running. Once the load is done, switch to the Management workstation and login. Click Source tab and then click the Refresh Services ( ) button to see if the project is loaded on the server. Select the new project and click the Deploy ( ) button. Once the service is successfully deployed, you can see the text file generated under the C:\Maca\Temp, as shown in Figure 237.

Figure 238.First result from GET method

The code that we pasted executed the GET method and got the result that is saved as a text file. Now, let's take a look at the code line by line. And we will explain what each line actually does. But before we move on, **please stop the service** or **undeploy** to stop creating the text files.

GET method code explanation

The first line you see is how we created the HttpClient instance, which is actually executes the HTTP request. Instead of direct instantiation, we used _httpClientFactory variable to create it.

```
using HttpClient httpClient = _httpClientFactory.CreateClient();
```

IHttpClientFactory

Maca uses the HttpClient Factory pattern, i.e., the **IHttpClientFactory**[38] object is injected to the MacaronService and you can use that in the code editor. The CreateClient() method will create a short-lived HttpClient object to execute HTTP methods. Since the HttpClient object implements IDisposable, it needs to be disposed when the underlying code is done. So, "**using**" keyword is required for the created HttpClient. Please keep this style when you start using the HttpClient.

Variable	Type	Usage Example	Detail
_httpClientFactory	IHttpClientFactory	_httpClientFactory.**CreateClient()**;	Manages HttpClient lifetime.

Note. Since the "GET todo list" service executes every minute and similar services may run concurrently, instantiating HttpClient directly with "new HttpClient()" will keep creating connections. Even if the HttpClient object gets disposed of with a using statement, the underlying socket will not be released immediately, which will lead to socket exhaustion in the end. So, please use _httpClientFactory.CreateClient(), as shown in the sample code every time.

GetAsync

Once you obtain the HttpClient, you can execute one of its HTTP request methods, such as GET, as shown below.

```
httpClient.BaseAddress = new Uri("https://jsonplaceholder.typicode.com");
```

One thing that's different from the tutorial site is the BaseAddress property assigned with https://jsonplaceholder.typicode.com as **Uri**[39]. Typically, the BaseAddress used to be defined when the HttpClient is registered[40] to the application in case of the factory pattern. As you will try later, we are not going to use only 1 fixed Uri. In many cases, the DNS could be changed at some point for variaous reasons, and there could be another web API to use with a different Uri while Maca is running. We will conver this in the later section. So, we will set BaseAdress once we obtain the HttpClient.

```
string routeQuery = "todos?userId=1&completed=false";
```

The second line is what we need to change in every request based on the requirements. The first part, "todos," is the **route** value for GET method. And the rest, starting with a question mark(?), are the **query strings** separated by an ampersand(&). Please check the tutorial site and the links related to the route stated in the previous sections. For now, we are requesting the web API server to return a list of Todo that's not completed for userId 1.

```
using HttpResponseMessage response = await httpClient.GetAsync(routeQuery);
```

Finally, we called the GetAsync with composed url and received the HttpResponseMessage[41] object. As you may have noticed, the HttpResponseMessage needs to be disposed of, and the "using" keyword is used. As we received the result of our request, we need to parse the content of it.

EnsureSuccessStatusCode

```
//response.EnsureSuccessStatusCode();
```

HttpResponseMessage that you received contains the information of the StatusCode[42]. The above line checks the value and throws an exception if the value is not success, e.g., 200 or Ok, and the rest of the code will not be executed. For now, we are not going to handle that exception, and comment the line to keep proceeding.

---

[38] https://learn.microsoft.com/en-us/dotnet/core/extensions/httpclient-factory
[39] https://learn.microsoft.com/en-us/dotnet/api/system.uri?view=net-9.0
[40] https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/use-httpclientfactory-to-implement-resilient-http-requests
[41] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.httpresponsemessage?view=net-9.0

Convert received HttpContent to a text

Inside of the received HttpResponseMessage, there is an HttpContent object that we need to parse and extract the data in most cases. Since the request and response use JSON (JavaScript Object Notation) for a data exchange, serialization and deserialization are required. In our case, we are receiving serialized JSON and thus a deserialization is required to read the content. For now, we are not going to do so. Instead, we will see what is in there and what we can do with that. And the JSON will be covered later in detail.

```
var jsonString = await response.Content.ReadAsStringAsync();
```

The ReadAsStringAsync() method will return the content's data as a human readable string, in this case as a JSON format. We will cover the detail of the content shortly.

Note. We will use the term "**content**" from now on for the HttpContent type objects. As you will see later, the content could be the objects, such as **JsonContent**[43], **StreamContent**[44], or **MultipartFormDataContent**[45], that inherit the HttpContent object.

Save the received content as a text file

As we extracted the content as a string, we can save that as a text file to open and see later.

```
string path = "C:\\Maca\\Temp";
if(!Directory.Exists(path))
   Directory.CreateDirectory(path);
File.WriteAllText(Path.Combine(path,DateTime.Now.ToString("yyyyMMddHHmmss"))+".txt", jsonString);
```

The above code will create a text file, as shown in Figure 238.

Maca's temporary file location

As the sample code will create .txt files, we need a local directory to save them. Maca will create a default directory for saving tutorial-related files, as shown below.

C:\Maca\Temp

Note. Throughout this tutorial, we will need some other locations to save sample and/or temporary files. To support that, the above folder will be used as a base directory to create other subfolders, such as "C:\Maca\Temp\Test\Files."

Review the received HttpContent

Now we are ready to see what we recived from the server. Go to C:\Maca\Temp folder and open the txt file.

---

[42] https://learn.microsoft.com/en-us/dotnet/api/system.net.httpstatuscode?view=net-9.0
[43] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.json.jsoncontent?view=net-9.0
[44] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.streamcontent?view=net-9.0
[45] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.multipartformdatacontent?view=net-9.0

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
  },
  {
    "userId": 1,
    "id": 2,
    "title": "quis ut nam facilis et officia qui",
    "completed": false
  },
  {
    "userId": 1,
    "id": 3,
    "title": "fugiat veniam minus",
    "completed": false
  },
  {
```


Figure 239.Received JsonContent

As we executed the GET method, the expected result is records, i.e., an array, of the data if exist. If you remember what we defined in the routeQuery, "todos?userId=1&completed=false," you will notice that the above result is filtered one based on that query. If we change the query to "todos?userId=1&**completed=true**" and redeploy the service, we will get the completed result only. For now, we simply dumped the HttpContent data to a text file. What we can do now with that is review what the data look like. In order to implement the client's requirements, we need to parse the data and do a job accordingly. In our case, especially the 2nd requirement, we will receive the result containing PDF data. So, we need to parse the result to extract PDF data to save that as a PDF file.

But before that, we need to keep testing various endpoints, which means we have to keep repeating the following: update the code, load the service, and deploy the service to verify the result. This might be a necessary procedure, but in the case of a simple test, such as checking if the API server is up and running, this could be tedious work. Or we may need some sample results from the server to use them as a resource or as a quick reference for the data parsing without service deployment. To support those situations, Damia provides a simple UI that executes HTTP methods and shows the returned results on the fly.

Introduction to the Web API client UI

If you have some experience with web API consumption, you may hear of the Postman[46] that sends HTTP requests, such as GET or POST, to the web API server. Since the Macaron Service is designed to consume web API services, it is necessary to execute HTTP requests and verify the response on the Damia side as well. To support that, the Web API client UI provides the following key features:

- Executes an HTTP request (GET, POST, PUT, and DELETE) and shows the response
- Generates C# code based on the selected HTTP request and insert it into the code editor

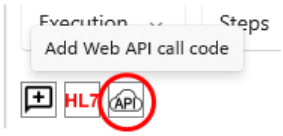
This very basic but configurable feature will help your service development. Throughout this tutorial, the features will be explained with examples. Now, let's take a look at the UI.

Open Web API client UI

Note. Before we proceed, **plese delete all code in the code editor**. The following steps will produce a new code that could be inserted into the part of the existing code. To avoid any code break or unexpected result, please make the code editor empty.

On the code editor, there is an API button with "Add Web API call code" right next to the HL7 button.

[46] https://www.postman.com/



Click the button, and the Web API client will pop up, as shown in Figure 240.

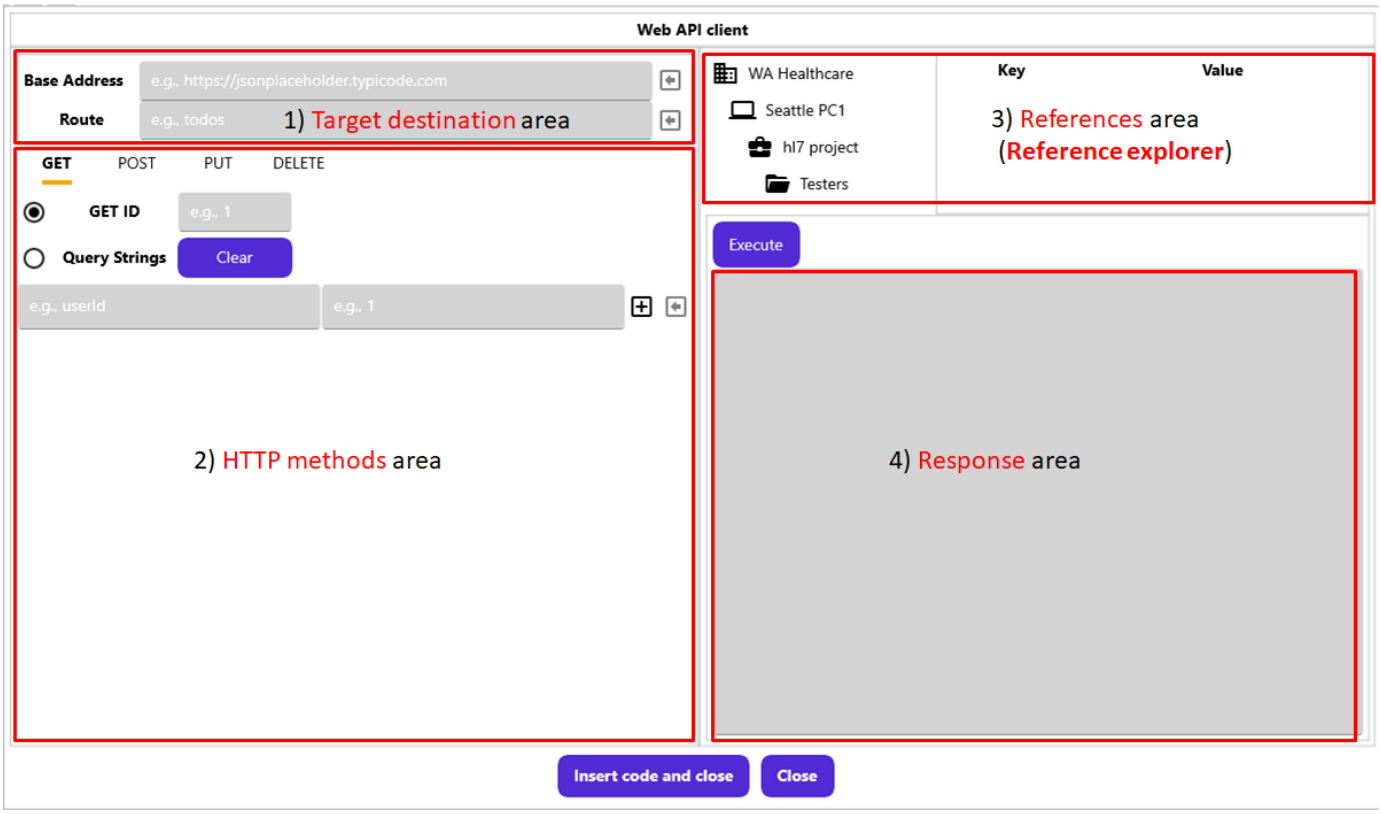


Figure 240.Web API client UI

As you see, there are 4 main areas that we can take advantage of: 1) Target destination, 2) HTTP methods, 3) References (Reference explorer), and 4) Response. Each part will be explained based on the previously used HTTP GET code in Figure 237. So, please check the code again.

Setup the endpoint

In order to call the GetAsync method, we composed the endpoint, which was a combination of base address, route, and query string that could be redefined like below.

https://jsonplaceholder.typicode.com/todos?userId=1&completed=true

As you will need to implement various HTTP requests, the final work that you need to do for each request is composing the endpoint based on the requirements. In the Web API client, we will use 2 areas, 1) Target destionation and 2) HTTP methods, to compose that endpoint.

Setup the target destination

Figure 241.Target destination input area

For every web API request, base address and route are required. Please type both values, or copy from the code that we used above and paste.

Setup the HTTP method

As you see in Figure 242, there are four HTTP method tabs, including the GET method that we will implement. When you open the UI, the GET tab is selected by default. For now, we are working on the GET method, so keep this as is. And we will try other tabs later.

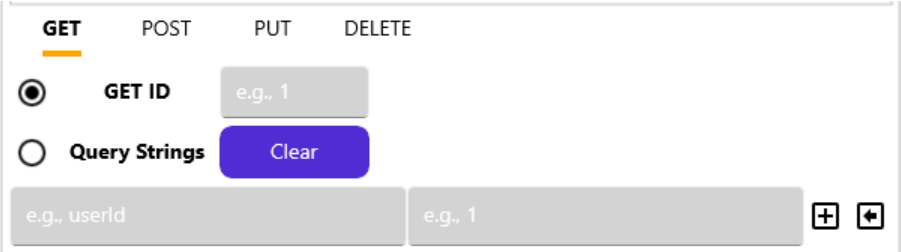


Figure 242.HTTP GET parameter input area

Execute the HTTP GET

Right above the Response area, there is an "Execute" button. Click that button, and the result will look like, as shown in Figure 243.

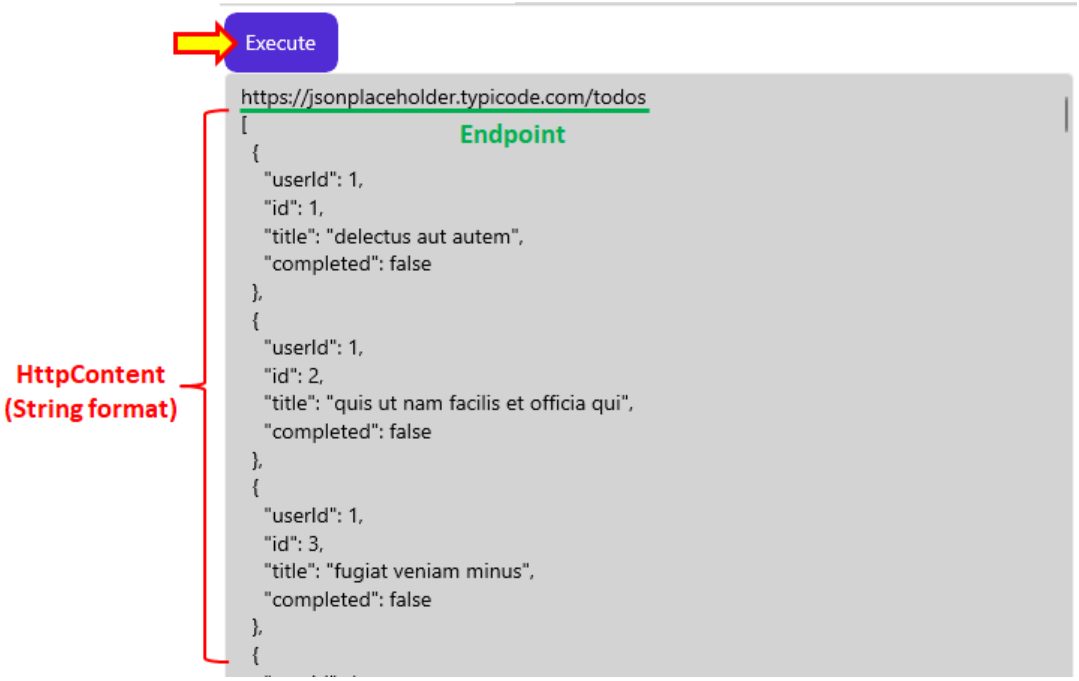


Figure 243.HTTP method execution and response

The response area shows two parts. **The endpoint** and the string representation of the received **HttpContent**. The endpoint will show you what request was submitted to the web API server, and the below will show what was returned as a result. So far, we didn't add any filtering data in our request. So the result will be the entire data in the server, which could be huge.

Note. The response area is limited to show 10,000 characters. Any huge data greater than that will be displayed partly.

Request filtered HTTP GET

So far, we executed a very initial HTTP request that gets all data from the web API server. This could be ok, but we need to filter out to get a more specific result from the server in most cases.

Filter with a specific id

A typical web API GET method could be requested with a specific id defined in the method in a format of "[route]/[value]." One of the usecases is using the identifiable id, such as system-generated user id for example. The tutorial site uses the id property, which is

a part of the Todo, as a filtering option. Since the tutorial set the value as "**todos/3**," we can use the value **3** in the GET ID section and execute, as shown in Figure 244.

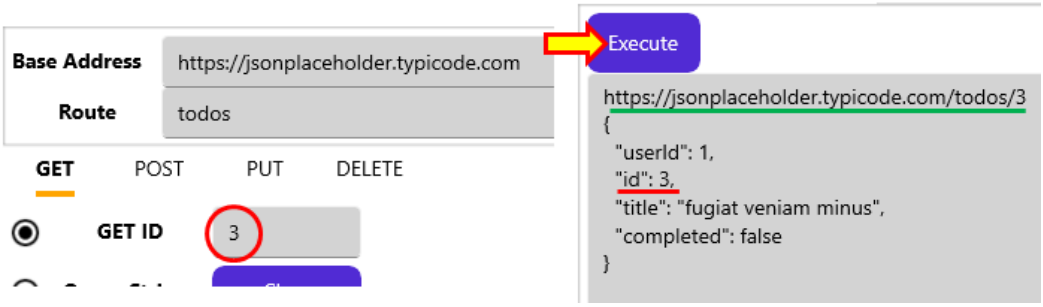


Figure 244.HTTP GET with id 3

You can see the endpoint is changed with "/3" value at the end. And the result shows the only 1 todo item with "id": 3. You can try more ids, such as 1 or 100 to see the result.

Note. The radio button name "GET ID" does not mean id in every case, although that usually is a system-generated number that increases by 1. The value followed by "[route]/" depends on the value defined in the web API. If the server defined the value to check products, then the "GET ID" could be interpreted as "GET PRODUCT ID." Likewise, if the endpoint is targeting the users, then the "GET ID" could mean "GET USER ID." So, please check the given web API specifications and set that accordingly.

Filter with query strings

Although the above request returns a specific data, we usually need more customized requests to get a specific result. As we used the *userId* and *completed* keys in the sample code to get the related result, we can define them in the query string area in the UI. Click the Query Strings radiobutton, as shown in Figure 245.

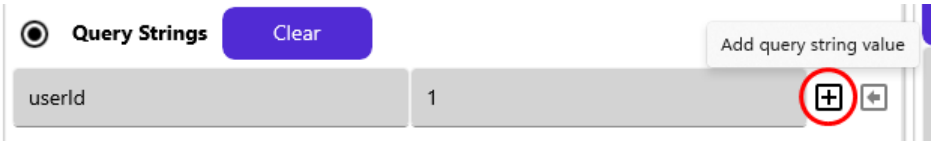


Figure 245.Add Query String

Right below the radio button, there are three input areas: key entry, value entry, and the add button. As we used two key values, *userId* and *completed*, to filter the result, we can type and add them here. Please type "userId" in the key entry and "1" in the value entry, then click the add button. Type "completed" and "false," then click the add button. Once all values are added, click the "Execute" button, as shown in Figure 246.

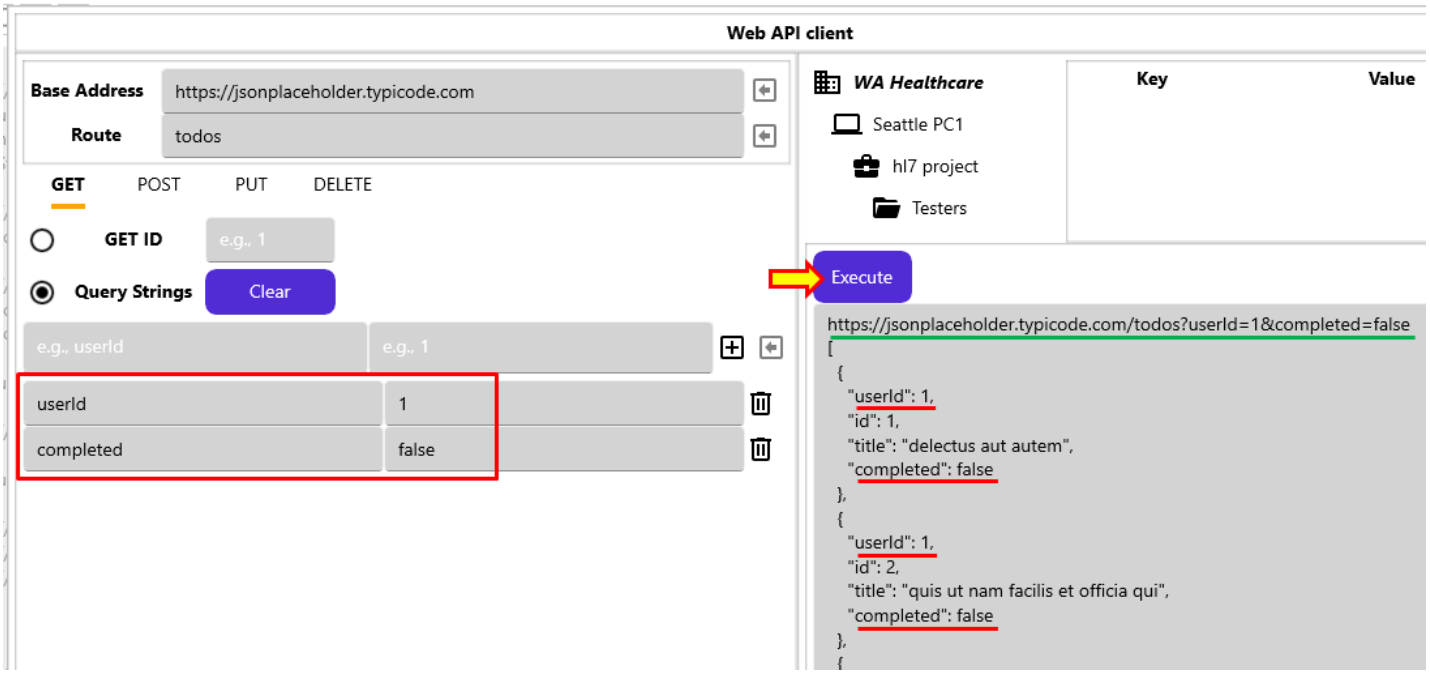


Figure 246.Execute HTTP GET with query strings

Finally, the endpoint in the response area is exactly the same as what we defined in the "Setup the endpoint" section. This shows what request was submitted to the web API server and what was returned as a result. As you noticed, the content part is what we saved to the text file in the earlier sections.

Note. If you want to check other Query String's value, you can directly modify added query strings in red box, as shown in Figure 246. For example, you can modify "false" value for the completed to "true" and then cllck the Execute button. Then you will see the result is only for the completed is true.

As we tested so far, we actually get the result without using the servcie deployment. By taking this approach, you can experdite the service development. You can tweak the request by adding or removing query strings to check the corresponding results.

Note. For removing a query string, you can click the delete button (🗑) next to the added query string. Or you can remove entire query stirngs by clicking "**Clear**" button next to the Query String radio button.

Insert Web API call code

The Web API client is quite handy to test HTTP methods on the fly. But just executing HTTP methods will not be enough in developing services because there are plenty of rich platforms in testing web APIs. To expedite the development, Damia provides Web API call code generation feature based on the configuration in the Web API client UI. Once you configure the endpoint and click the "Insert code and close" button, the corresponding code will be inserted into the code editor.

As we executed the GET method with query strings, we can keep this configuration to see if we have the same result compared to the previous hard-coded one. So, click the "Insert code and close" button.

```
// Creates short-lived HttpClient instance
using HttpClient httpClient = _httpClientFactory.CreateClient();
httpClient.BaseAddress = new Uri("https://jsonplaceholder.typicode.com");

StringBuilder routeQuery = new();
// Route
routeQuery.Append("todos");

// Query Strings: Replace the value with a variable, e.g., ?userId={_id}
routeQuery.Append($"?userId=1");
routeQuery.Append($"&completed=false");

using HttpResponseMessage response = await httpClient.GetAsync( routeQuery.ToString() );

//response.EnsureSuccessStatusCode();

using JsonDocument jsonDoc = JsonDocument.Parse( await response.Content.ReadAsStringAsync() );

// Based on the jsonDoc.RootElement.ValueKind, parse Object or Array accordingly
// jsonDoc.RootElement.GetProperty("userId") ← if { } returned
// foreach (JsonElement element in jsonDoc.RootElement.EnumerateArray()) ← if [ ] returned
```

Figure 247.Auto-generated HTTP GET code

As you see in Figure 247, several lines of code were inserted into the code editor. Let's take a look at that one by one.

Compared to the hard-coded one, the first part that you will notice is that the routeQuery was divided into two parts: Route and Query Strings. Instead of a one-line string like "todos?userId=1&completed=false," multi-line code using StringBuilder will be more readable and easy to modify with the variables. So, you can easily identify each query string and modify it as necessary, like "?userId=1" to "?userId={_id}," if _id is a variable or a passed parameter. We can add more Query Strings on the Web API client UI and configure them here. Or you can directly modify them here if that's more comfortable. We will cover that more in the later section, but for now, keep this value as is.

Note. For the variables in the routeQuery lines, string interpolation[47] was applied to make sure the string value accepts the variable.

The second part that's new here is the JsonDocument.

JSON Deserialization

Before we check the JsonDocument, let's think about the HTTP communication. Since the HTTP request, e.g., the POST case, and response require JavaScript Object Notation (JSON) serialization and deserialization to exchange the data with the web API server, we need to deserialize the received serialized JSON to a strongly typed C# object in order to read the values. In the previous example, we just dumped the data to the text file as a human readable JSON. But now, we are going to conver that to a C# object to read each value one by one.

If you look at the tutorial site, you will notice that there is an extension method, **GetFromJsonAsync**, of HttpClient that converts a JSON to a C# object, in that case Todo class, like below.

```
public record class Todo(…
```

By using the Todo class, the HTTP GET and the conversion was simplified with a line of code. And we could get the readable List<Todo> C# object like below.

```
var todos = await httpClient.GetFromJsonAsync<List<Todo>>("todos?userId=1&completed=false");
```

That is a quite handy way of handling all steps, but there is a limitation in Maca to recognize the classe by design.

[47] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated

In order to use the Todo class, i.e., be referenced in the MacaronService, importing the DLL that contains the Todo class is required. If the web API project uses the separate DLL library, you can use that. However, if the class in the DLL requires another DLL, you have to import that as well, which could be an annoying process if the DLL is updated later. In addition, there is a concern that other unused classes are also exposed. Otherwise, you have to create and maintain another ad-hoc class library project that contains only the Todo class. Although these are possible options, it is not a simple process in order to use just a few classes. Instead, Maca uses the JsonDocument as an alternative to convert (deserialize) the serialized JSON.

JsonDocument[48]

JsonDocument's Parse static method converts the string value to a JsonDocument object. As we converted the returnd HttpContent to a string using ReadAsStringAsync(), we can reuse that to generate string value for the Parse method. Since the JsonDocument implements IDisposable, we also used "using" keyword to make sure the release of the used resources.

Note. There is another object, JsonNode[49], that can be converted to. But the JsonDocument provides a faster access time in reading each element, so we recommend using JsonDocument for the conversion.

Read JsonDocument

Once we obtain the JsonDocument object, we need to check which type was returned, i.e., a single object { } or an array object [ ]. As we saw dumped value in the txt file, the value we received was an array type that starts with an opened square-bracket, i.e., [. If we expect multiple record type data, the returned value will be [ ]. Or, if we requested the specific data, like GET ID is 1 case, we would receive curly brace data like { } based on the Jsonplaceholder web API. But instead of fixed code, we can identify them using the ValueKind property, which is JsonValueKind[50] enum. Now, we know we will receive an array or an object, we can configure the code like below example.

```
StringBuilder jsonString = new();
if (jsonDoc.RootElement.ValueKind is JsonValueKind.Object)
{
   // GET ID returns { }
   jsonString.Append("userId:" + jsonDoc.RootElement.GetProperty("userId"));
   jsonString.AppendLine(", completed:" + jsonDoc.RootElement.GetProperty("completed"));
}
else
{
   // Query Strings returns [ ]   JsonValueKind.Array
   foreach (JsonElement element in jsonDoc.RootElement.EnumerateArray())
   {
      jsonString.AppendLine(element.GetRawText());
      // jsonString.Append("userId:" + element.GetProperty("userId").GetInt32());
      // jsonString.AppendLine(", completed:" + element.GetProperty("completed").GetBoolean());
   }
}
```

Figure 248.Sample code identifies an object or array and processes it accordingly

Please note that JsonDocument always starts with RootElement, which is a JsonElement[51] struct. Although what we received was a JsonDocument, what we will do most is check the JsonElement. And we can access each element by using the EnumerateArray() method in case of an array. As you saw in the example, we accessed each property's value by using GetRawTest() and GetProperty(). For now, we use the GetRawText() only to see if what the result looks like compared to the previous service execution. You can copy above code and replace the last 3 commented line with it. Then you can reuse the code that writes the data to a text file like below.

```
string path = "C:\\Maca\\Temp";
```

[48] https://learn.microsoft.com/en-us/dotnet/api/system.text.json.jsondocument?view=net-9.0
[49] https://learn.microsoft.com/en-us/dotnet/api/system.text.json.nodes.jsonnode?view=net-9.0
[50] https://learn.microsoft.com/en-us/dotnet/api/system.text.json.jsonvaluekind?view=net-9.0
[51] https://learn.microsoft.com/en-us/dotnet/api/system.text.json.jsonelement?view=net-9.0

```
if(!Directory.Exists(path))
  Directory.CreateDirectory(path);

File.WriteAllText(Path.Combine(path,DateTime.Now.ToString("yyyyMMddHHmmss"))+".txt", jsonString.ToString());
```

Please note that we replaced the value part with **jsonString.ToString()**;

Finally, the Complete code will look like below.

```
// Creates short-lived HttpClient instance
using HttpClient httpClient = _httpClientFactory.CreateClient();
httpClient.BaseAddress = new Uri("https://jsonplaceholder.typicode.com");

StringBuilder routeQuery = new();
// Route
routeQuery.Append("todos");

// Query Strings: Replace the value with a variable, e.g., ?userId={_id}
routeQuery.Append($"?userId=1");
routeQuery.Append($"&completed=false");

using HttpResponseMessage response = await httpClient.GetAsync( routeQuery.ToString() );

//response.EnsureSuccessStatusCode();

using JsonDocument jsonDoc = JsonDocument.Parse( await response.Content.ReadAsStringAsync() );

StringBuilder jsonString = new();
if (jsonDoc.RootElement.ValueKind is JsonValueKind.Object)
{
    // GET ID returns { }
    jsonString.Append("userId:" + jsonDoc.RootElement.GetProperty("userId"));
    jsonString.AppendLine(", completed:" + jsonDoc.RootElement.GetProperty("completed"));
}
else
{
    // Query Strings returns [ ]   JsonValueKind.Array
    foreach (JsonElement element in jsonDoc.RootElement.EnumerateArray())
    {
        jsonString.AppendLine(element.GetRawText());
        //jsonString.Append("userId:" + element.GetProperty("userId").GetInt32());
        //jsonString.AppendLine(", completed:" + element.GetProperty("completed").GetBoolean());
    }
}

string path = "C:\\Maca\\Temp";
if(!Directory.Exists(path))
  Directory.CreateDirectory(path);

File.WriteAllText(Path.Combine(path,DateTime.Now.ToString("yyyyMMddHHmmss"))+".txt", jsonString.ToString());
```

Figure 249.HTTP GET sample code

Now, you can click the Verify( ) button to see if any error comes up. If there's no issue, load this to the server. If the Macaron server is not up and running, please make sure start the server. Then login to the server and deploy the service. **Once the service starts, please stop that to avoid further file creation**. Go to the C:\Maca\Temp folder. You will see the newly created txt file, as shown in Figure 250.

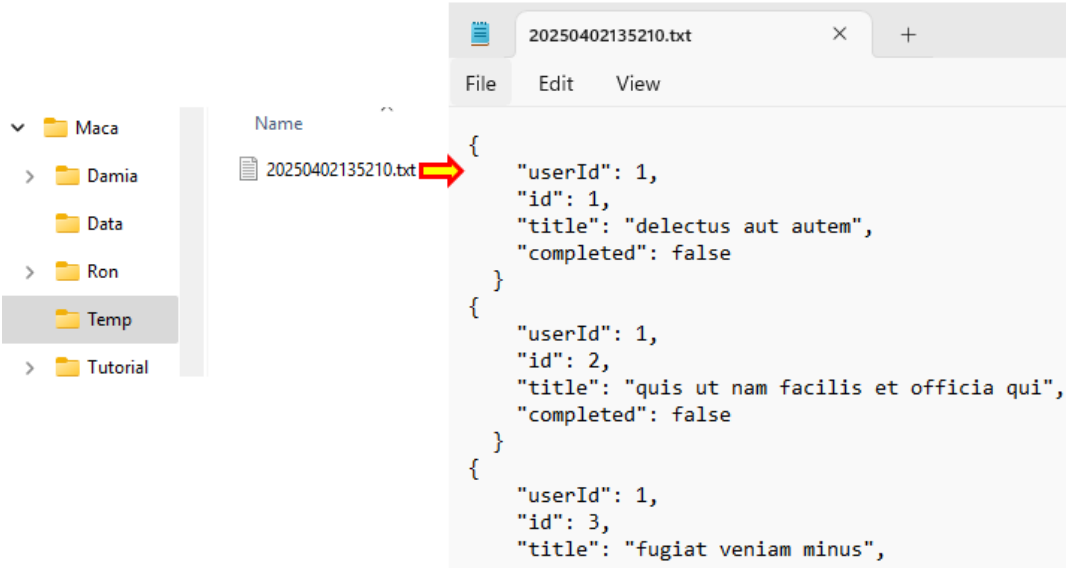


Figure 250.Created JsonContent text

When you open the file, you can see that there is no huge difference from the previous execution, except the text doesn't have [ ]. Although we simply captured the raw value of each element and saved it to the txt file, we learned what we can do with each element. Let's uncomment two lines that access the userId and completed properties and redeploy the service.

```
// Query Strings returns [ ]   JsonValueKind.Array
foreach (JsonElement element in jsonDoc.RootElement.EnumerateArray())
{
    //jsonString.AppendLine(element.GetRawText());
    jsonString.Append("userId:" + element.GetProperty("userId").GetInt32());
    jsonString.AppendLine(", completed:" + element.GetProperty("completed").GetBoolean());
}
```

Figure 251.Getting property values

Then the new deployment will generate the following txt file like below, as shown in Figure 252.

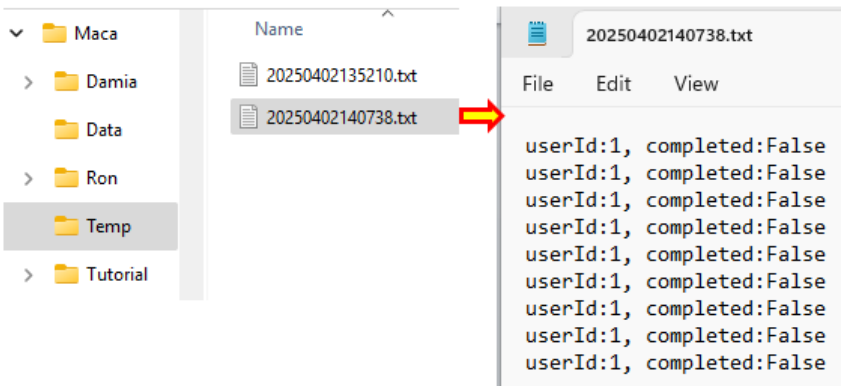


Figure 252.Saved property values

Although we didn't get the distinguishable properties like id or title, we now know how we can access each element and its properties.

So far, we used the Web API client to generate HTTP GET method call code and used JsonDocument to read the returned content. With the above steps, you can play more on the HTTP GET with the UI. As you may realize, this technique can be used to implement the 2nd requirement that downloads the pdf file from the web API server. We will cover that in the later section. But for now, let's try other HTTP methods using the Web API client, especially the HTTP POST method.

Execute the HTTP POST

Before we implement the service, let's take a quick look at the HTTP POST.

HTTP POST behavior

As you saw in the tutorial site, the HTTP POST is little different from the HTTP GET in terms of the data to exchange. The GET method submits a request with the id or query strings because the main purpose of the method is to get the requested data. On the contrary, the POST method is to request the server to create new data with submtting data. So, we need to define the object followed by the specification that the web API server requires.

Note. The object to submit will be the HttpContent[52] type. In many POST cases, we will submit one of the derived clasess, such as ByteArrayContent[53], JsonContent[54], StringContent[55], StreamContent[56], and MultipartFormDataContent[57].

JSON Serialization

In the Json Deserialization section, we saw that we have to convert(deserialize) the received JSON to the C# object, i.e., the JsonDocument, in case of HTTP GET. But at this time, we have to serialize the C# object to JSON in order to submit that to the web API server with HTTP POST. And we have a similar issue with HTTP POST in terms of JSON serialization when using the HTTP extension method PostAsJsonAsync that uses Todo class. As we explained that we are not using DLL import option, we will not use the extension method as well. Instead, we will use the PostAsync method and use the anonymous object for the JSON serialization.

Anonymous Object[58]

If you look at the sample code of the POST method on the tutorial site, you can identify the "JsonSerializer.Serialize" part that serializes the anonymous object to a string that is used to create the StringContent. Then the content is used as the 3rd argument of the PostAsync method. And we will use the internal part, which is an anonymous object in the red box, for the serialization, as shown in Figure 253.

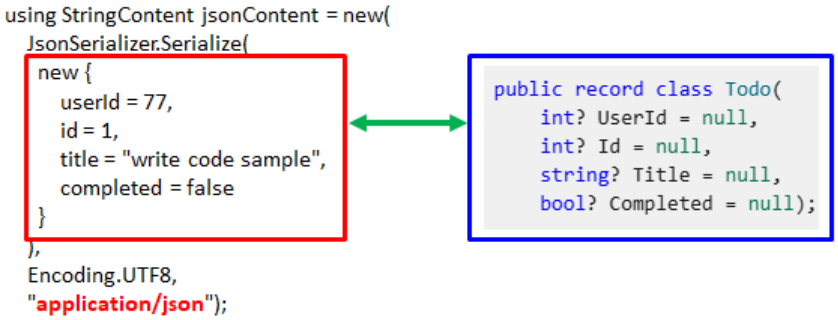


Figure 253.Anonymous object and Todo record class

Once you look at the anonymous object's properties(userId, id, title, and completed), you will notice that the structure is exactly the same as the Todo class in the blue box. In other words, no matter what object we submit, there is no change in the baseline that we have to provie all data that the web API server requires. The only difference is the syntax.

Note. The HttpClient tutorial site's examples use camel case (**userId**) for the property in the anonymous object for quick mapping with the JSON property (**"userId"**). Followed by the naming rules[59] that use Pascal case for the properties, **UserId** would be morea

[52] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.httpcontent?view=net-9.0
[53] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.bytearraycontent?view=net-9.0
[54] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.json.jsoncontent?view=net-9.0
[55] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.stringcontent?view=net-9.0
[56] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.streamcontent?view=net-9.0
[57] https://learn.microsoft.com/en-us/dotnet/api/system.net.http.multipartformdatacontent?view=net-9.0
[58] https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/anonymous-types

reasonable for the given anonymous object's property. But both are acceptable in most cases, so please choose whichever is comfortable for you to implement. And we will keep using camel case in this tutorial.

By following this format, we can submit the anonymous type object without using a specfic C# object, in this case Todo record class. And based on this example, we understand that the web API server requires a JSON-type content, i.e., JsonContent. If you look at the 3rd argument ("**application/json**") of the Serialize method, you will notice that the HttpContent will be submitted as a JSON **Media Type**[60](formerly known as MIME types). So, we will submit the anonymous object as JsonContent to the web API server instead of StringContent. For the list of the applications including json Media Type, please check https://www.iana.org/assignments/media-types/media-types.xhtml#application.

Note. It is common to submit StringContent containing JSON data with "application/json" media type to the server because it is still acceptable by the server. But sometimes it may not be acceptable and the unexpected result will be returned as a result based on the server side configuration. For the JSON type data submission, it is more reliable to submit JsonContent over StringContent.

Create a POST service

So far, we defined that we are going to use the anonymous object for the HTTP POST and convert(serialize) it to the JsonContent. Lastly, we defined the content will be "application/json" media type. But before we try that method with an existing service, create a new service named "POST todo list" as CodeRepeater first. (You can name it "POST todo item" instead. So, feel free to name it what better fits to you.) We can create it with other service types later but set it as CodeRepeater for the simple use. You can clone the "GET todo list" service and rename it as an option.

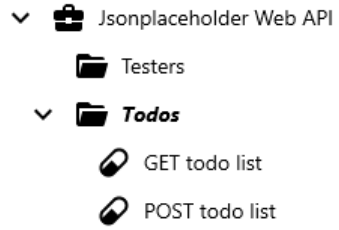


Figure 254.Added POST todo list on the project explorer.

Add References for the web API

In the previous steps, we set the GET todo list service to call the HTTP GET method. And we typed the base address and route manually like below Figure 255.

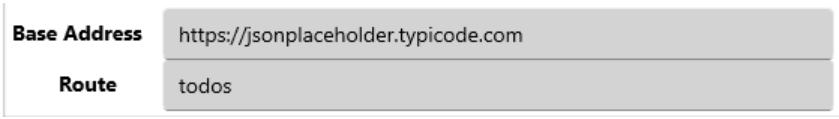


Figure 255.Manually typed endpoint information

What we are going to do is assign them to the References in the Project and the Group and then associate them with the service. First of all, select the "Jsonplaceholder Web API" node on the project explorer. When you see the detail page, you can type the key and value with "baseAddress" and "https://jsonplaceholder.typicode.com" and click the Add button, as shown in Figure 256. **Please save the service** before you move on to the next step.

[59] https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names
[60] https://www.iana.org/assignments/media-types/media-types.xhtml

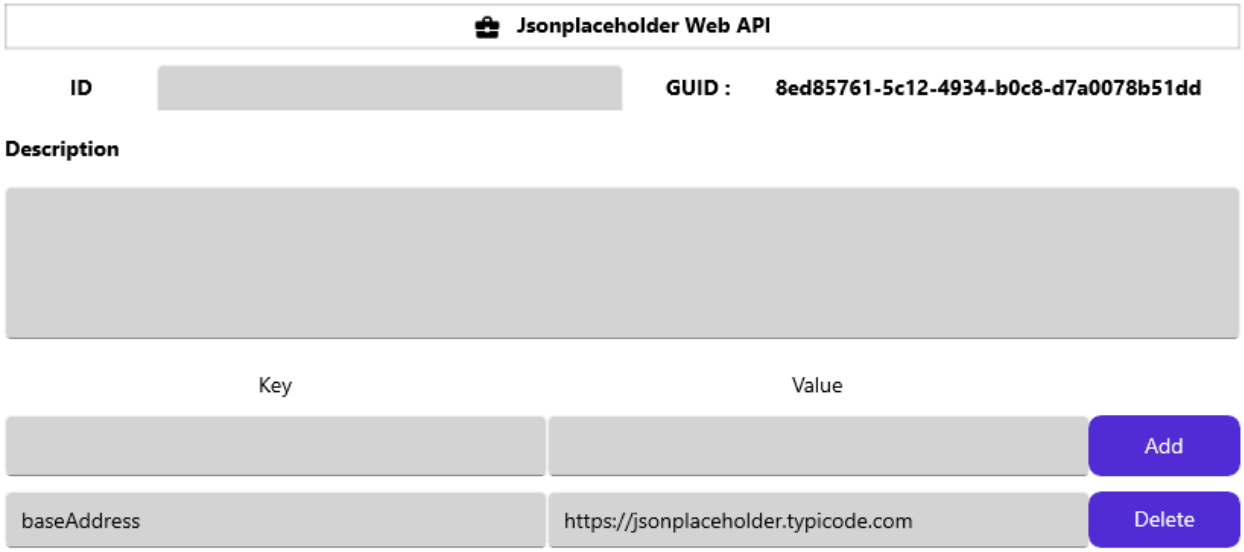


Figure 256.Added baseAddress Reference in the project

Then select "Todos" group and type the key and value with "route" and "todos," as shown in Figure 257.

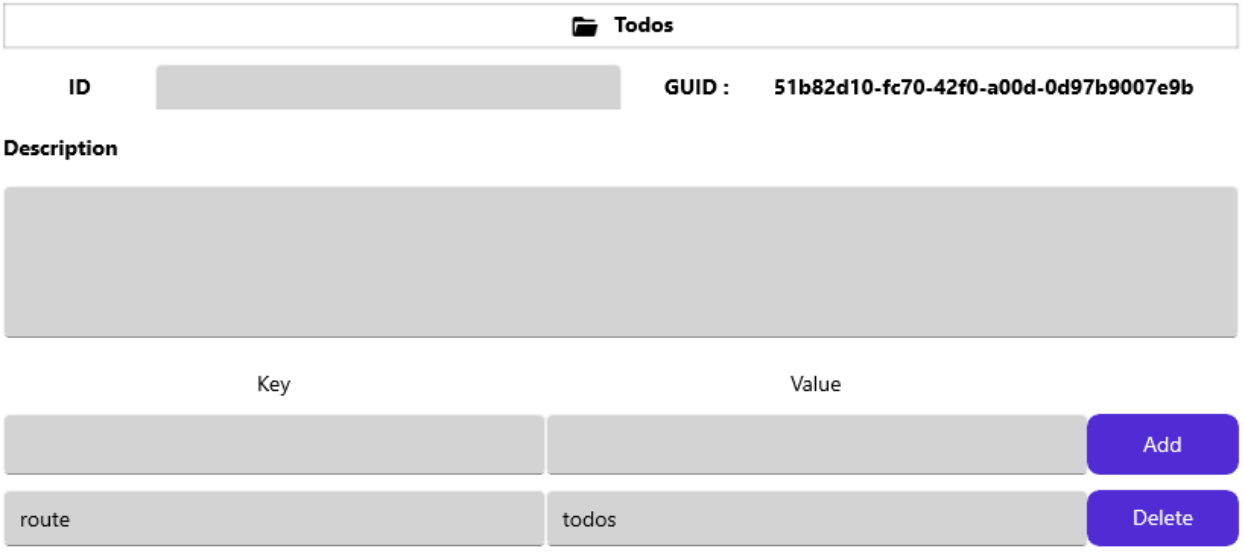


Figure 257.Added route Reference in the GroupOnce

Once you add the values, **save the service**.

As we saved the web API related info to the Project and Group, we will use them in the newly created service as well as in the Web API client UI.

Reference Association

Please select the POST todo list service node on the project explorer, and go to the Execution > Steps view. At this time, we are not going to directly type C# code in the code editor. We will use Web API client to generate the code and insert that into here. Click the API button ( ) to open the UI. Once the UI shows up, click the project node and group node to see what we just assigned, as shown in Figure 258.

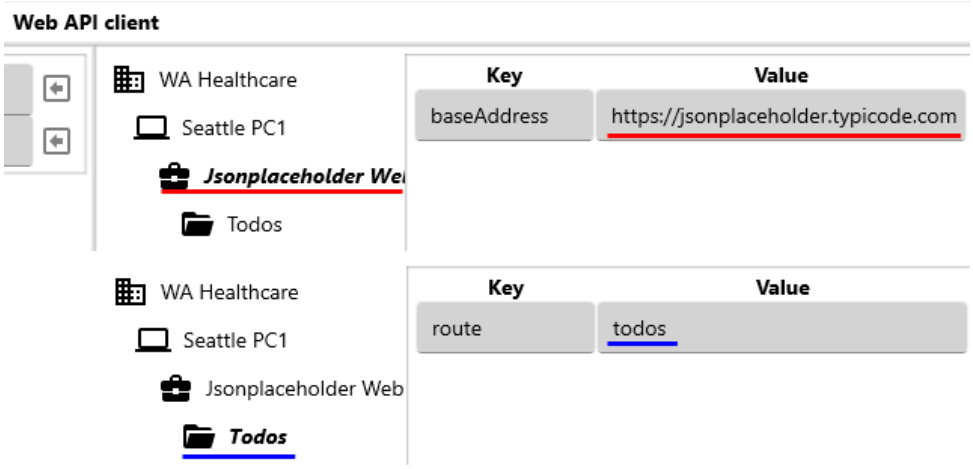


Figure 258.Added References in the Project and Group.

As we assigned, the project has Jsonplaceholder Web API's base address information, and the group has route information. Now, let's associate them with the current service.

Click the value of the baseAddress and the arrow button next to the base address input entry will be enabled. Click the arrow button, as shown in Figure 259.

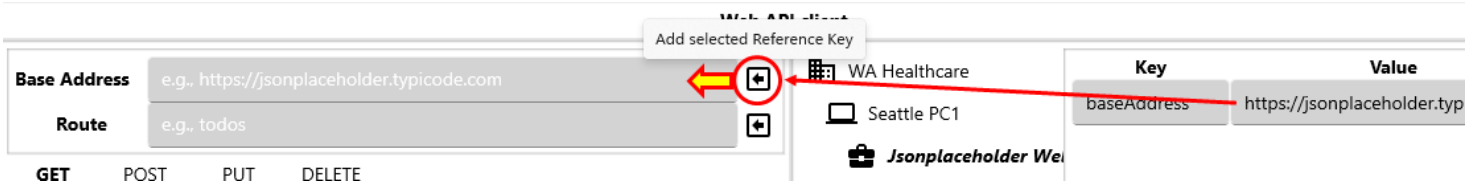


Figure 259.Add project's baseAddress key to the Base Address area

Once you add the key, click the Todos group and then select the route value. Then click the arrow button next to the Route. When both keys are added, it will look like below, as shown in Figure 260.

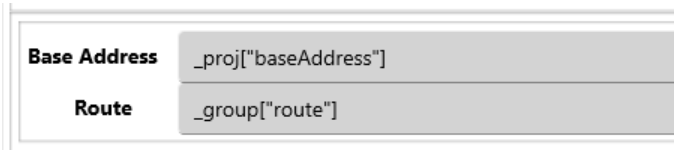


Figure 260.Added Reference keys

Although the service is for the POST method, we may try GET again just to see how these two associated References work. As we tried before, let's execute GET method. At this time, set the GET id with 1. And then click the Execute button. Then the result will look like below, as shown in Figure 261.

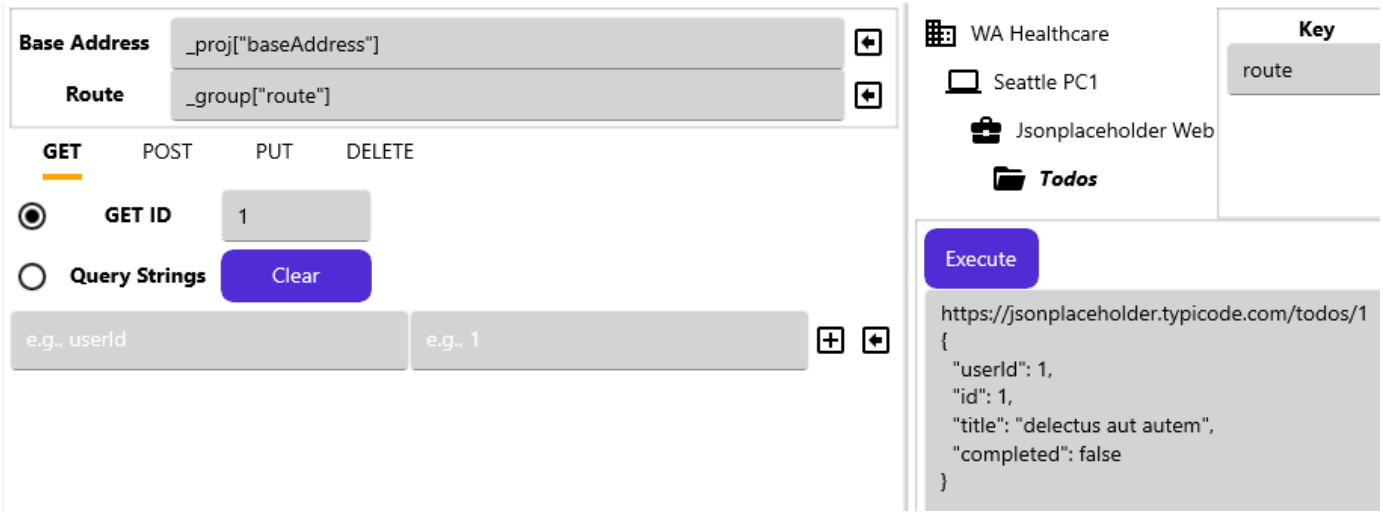


Figure 261.Execute HTTP GET with Reference keys

As the response area shows, we could successfully execute HTTP GET and received the expected result, which is id 1.

By using this way, you no longer need to type base address and route in case of the new servcie creation. Even this will make your service flexible when the base address or route are changed, because one value change in the project or group will affect all underlying services. In addition, you don't have to redeploy the services because Maca uses short-lived HttpClient object that uses entire endpoint everytime, the changed value will be affected while the services are running. Now let's move on to the POST. Click the POST tab right next to the GET tab.

Configure the POST in the Web API client

If you click the POST tab, you will see the two main sections: HttpContent picker and Anonymous object input panel.

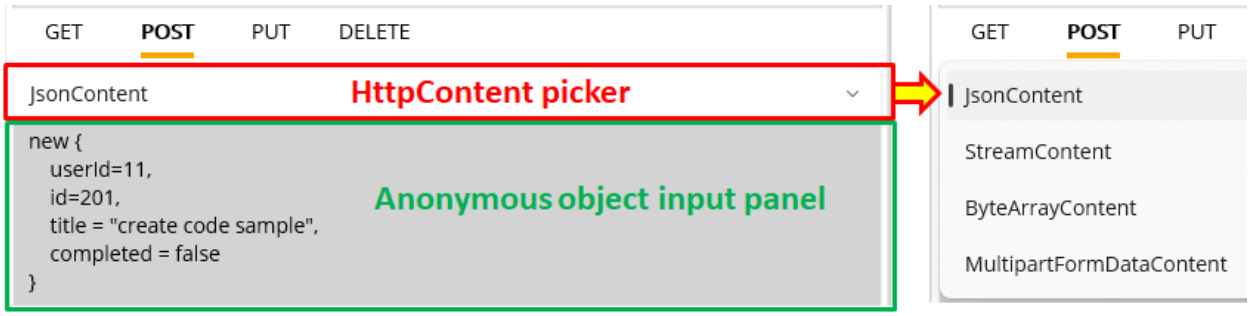


Figure 262.POST method input area

The initial selection is JsonContent and you can click the picker to change to one of the lists. If you pick the other one, such as StreamContent, the below section will be chagned accordingly. For now, we are going to submit the anonymous object as is. Without changing anything in the panel, click the Execute button.

Note. Anonymous object input panel is populated with the Jsonplaceholder's sample value. You can freely modify to the other format using that. For the complex object, please refer to the anonymous type tutorial site.

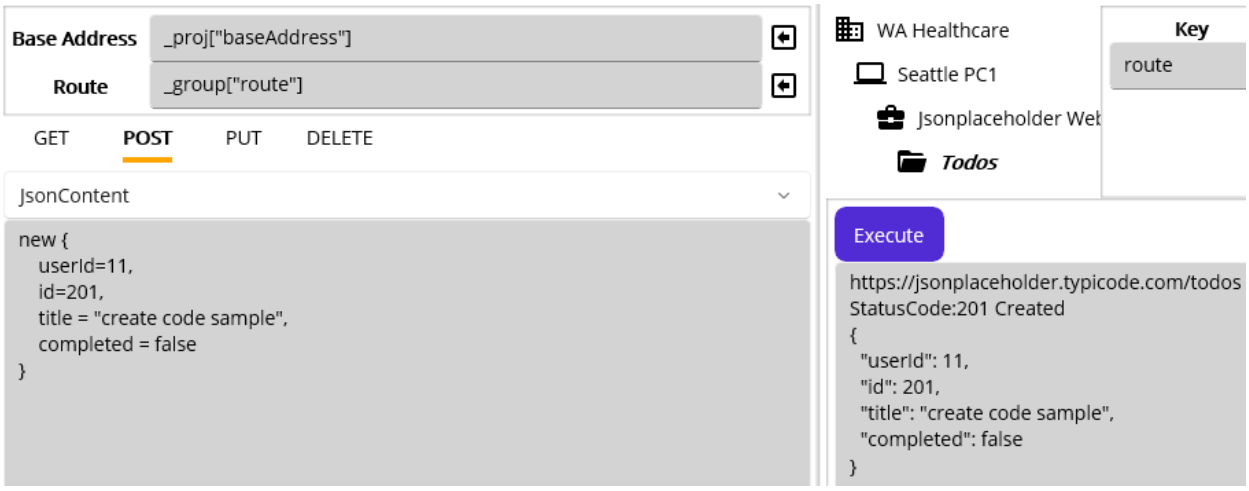


Figure 263.Initial execution with a default JsonContent.

As you see in Figure 263, we submitted the JsonContent to the endpoint "https://jsonplaceholder.typicode.com/todos" and received the result with the StatusCode 201. And the server returned the corresponding json content.

Note. The Jsonplaceholder Web API doesn't actually create, update, or delete the data with submitted values. In case of the POST, the server will tolerate the submitted object with missing property and return StatusCode 201 as a result. For example, if you delete userId and id properties and then click the Execute button, you will see the StatusCode 201 again. The site simply reacts based on the request, i.e., return values according to the submitted value. And the submittable object is limited to the JsonContent, i.e., if you submit the other content, such as StreamContent, the server will return unexpected result. So, we will use the site only for the purpose of practicing HTTP methods.

Insert the POST code

As we saw the result on the UI, let's insert the POST code into the code editor. Click "Insert code and close" button. Now, you will see the inserted code that looks similar but somewhat different in the HttpContent part, as shown in Figure 264.

```
// Creates short-lived HttpClient instance
using HttpClient httpClient = _httpClientFactory.CreateClient();
httpClient.BaseAddress = new Uri(_proj["baseAddress"]);

// Route
string route = _group["route"];

var jsonPayload = new {
    userId=11,
    id=201,
    title = "create code sample",
    completed = false
};
using JsonContent httpContent = JsonContent.Create( jsonPayload );

using HttpResponseMessage response = await httpClient.PostAsync( route, httpContent );

//response.EnsureSuccessStatusCode();

using JsonDocument jsonDoc = JsonDocument.Parse( await response.Content.ReadAsStringAsync() );

// Use response.StatusCode and jsonDoc.RootElement
```

Figure 264.Auto-generated HTTP POST code

If you look at the code, the first thing we have to look at carefully is the anonymous object part marked with a right brace in red. As we decided not to use the Todo class, we used the anonymous object as an alternative. But the value assigned to each property is hard-coded, and thus the result after each execution will be the same. Since this auto-generated code is a kind of code template, it is

required to modify properties with the local variables and/or passed parameters accordingly to implement the requirements. We will cover that topic a little later and focus on the fixed value for now.

The 2nd part, which is related to the serialization, is the JsonContent instantiation. The static method Create serialized the anonymous object "jsonPayload" to the JsonContent.

Lastly, the JsonContent is assigned to the 2nd argument of the PostAsync method. And the Parse method converted(deserialized) returned HttpREsponseMessage to the JsonDocument.

As you saw before, EnsureSuccessStatusCode() is commented here to keep proceeding regardless of the StatusCode. So, we need to handle the statuc code in the following lines. In case you let Maca handles the exception, i.e., do not proceed the rest of the code and save the exception detail to the database, you can uncomment it. For now, we make the rest of the code the same as before, i.e., create a text file and dump all data to it along with the status code, as shown in below Figure 265.

```
// Use response.StatusCode and jsonDoc.RootElement
string path = "C:\\Maca\\Temp";
if(!Directory.Exists(path))
  Directory.CreateDirectory(path);

StringBuilder buffer = new();
buffer.AppendLine( $"{(int)response.StatusCode} : {response.StatusCode}" );
buffer.AppendLine( jsonDoc.RootElement.GetRawText() );

File.WriteAllText(Path.Combine(path,DateTime.Now.ToString("yyyyMMddHHmmss"))+".txt", buffer.ToString());
```

Figure 265.POST response saving code

You can copy the code and past it at the end of the line. If you want to handle the response as what we did in the GET example, you can reuese that and handle the returned JsonDocument accordingly. For this execution, we used response.StatusCode to add in the text file to see what's the Status. Then we used root emelent's GetRawText() method to dump the result data as is. With this code, let's execute the service to see what the result looks like. Click the **Veryfy** ( ) button. If there's no issue, load the service and deploy.

Figure 266.Downloaded result and the Web API client execution result

Once the service was started, stop it and go to the C:\Maca\Temp folder to see what was generated. When you open the text file, you can see the dumped data with the status code, as you see in Figure 266, If you compare it to what we saw in the Web API client UI, you will notice that there's no difference.

So far, we just tested the happy path with the valid input. Let's see what Maca will do when "EnsureSuccessStatusCode()" throws an exception in case a non-success status was returned from the web API server?

Basic Exception Handling with EnsureSuccessStatusCode()

Set EnsureSuccessStatusCode()

As we decided to use EnsureSuccessStatusCode(), please uncomment that line in the code editor.

```
using HttpResponseMessage response = await httpClient

response.EnsureSuccessStatusCode();

using JsonDocument jsonDoc = JsonDocument.Parse( awai
```

Figure 267.Uncommented response.EnsureSuccessStatuscode();

Click the **Verify** ( ) button and load the service. Once you redeploy the service, you will see a new text file was generated because there was no incorrect or invalid data in the request. At this time, **do not stop the service**, i.e., let the service keep generating the text files.

As you remember, we associated the base address and route values with the project's and group's References. Let's use that value to generate an exception. Since the service uses a short-lived HttpClient that gets the route value from the group's References every minute, we can change the route value in the group with an invalid one to generate an exception while the service is running.

Set an invalid route value

Now, what we will do is modify the route key's value stored in the Todos group's References with an incorrect value, "files," and deploy the group to see what will happen as a result. What we can expect with this configuration is the HTTP POST will not return a 201 status code, and no txt file will be generated. And the transaction will be saved as an error in the database by Maca.

Click the Todos group on the project explorer. **Update the route key's value to "files,"** as shown in Figure 268.

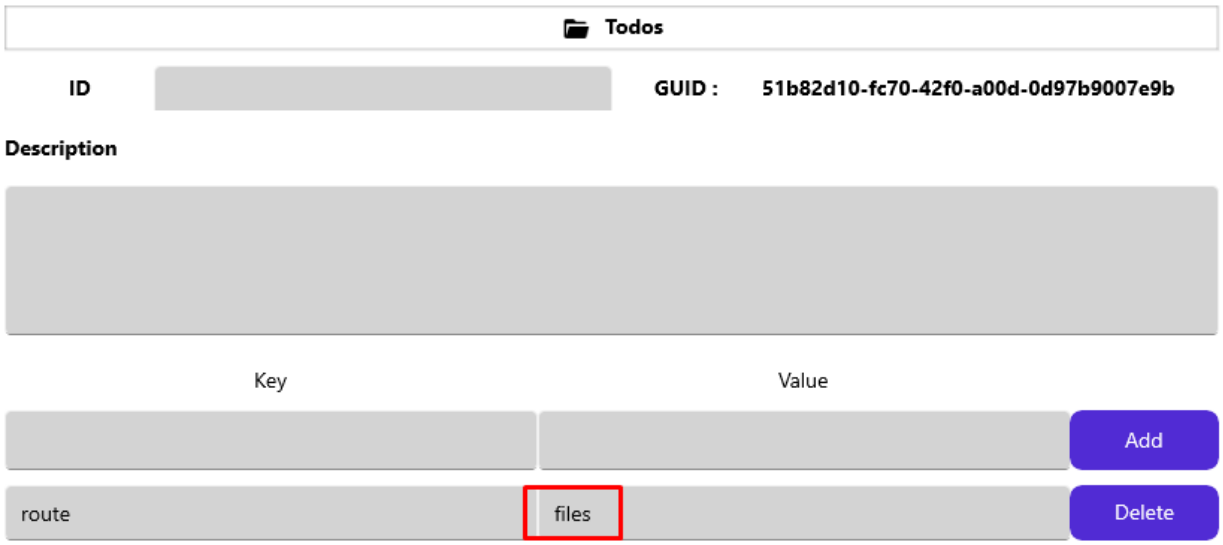


Figure 268.Update the route key's value to "files" in the Todos References

Once you update that value, please **click the Save** ( ) button to save. Then **click the Load** ( ) button to load the group. But at this time, we need to **load the group only**, **click No button**, as shown in Figure 269.

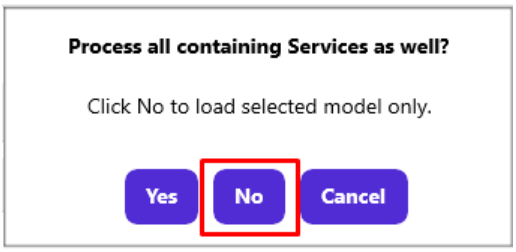


Figure 269.Load only the Todos group

Deploy modified group

Go to the Management workstation and click Source tab, then **select the Todos group node**. Click the **Deploy** ( ) button. As we will deploy the group only, click the No button, as shown in Figure 270.

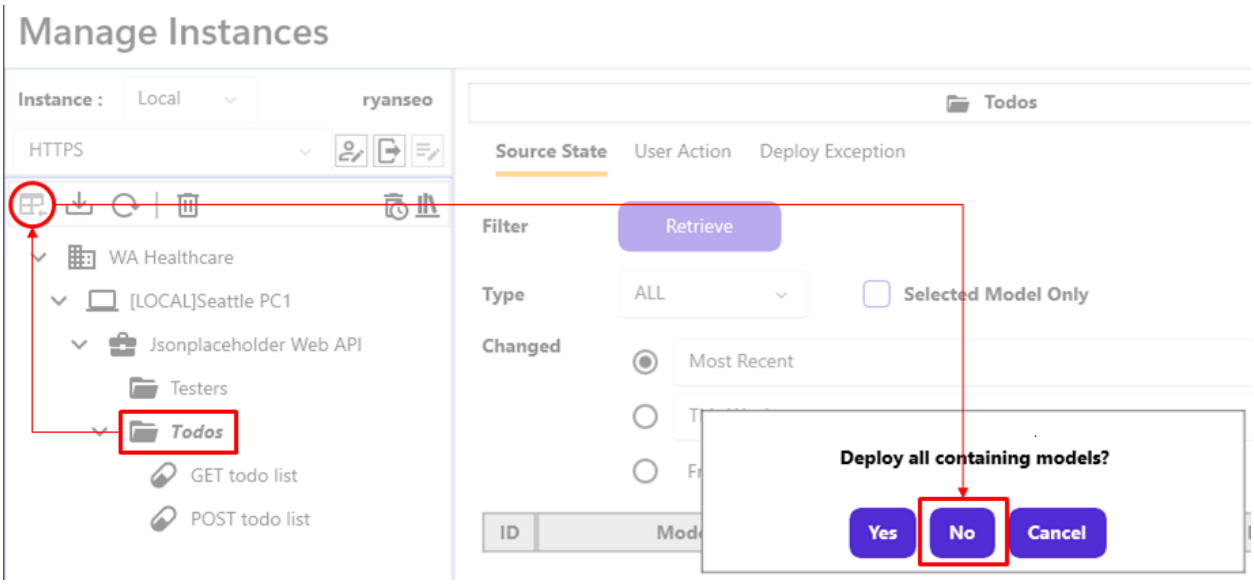


Figure 270.Deploy Todos group only

Retrieve the transaction history of the service

Since we didn't stop the POST todo list service, there will be a history of its transaction in the database. We use that to see what was happening after the group's route value was modified. Click the POST todo list service. On the Observe > Transaction, set "Today" and click the "Retrieve" button to retrieve the history of the service, as shown in Figure 271.

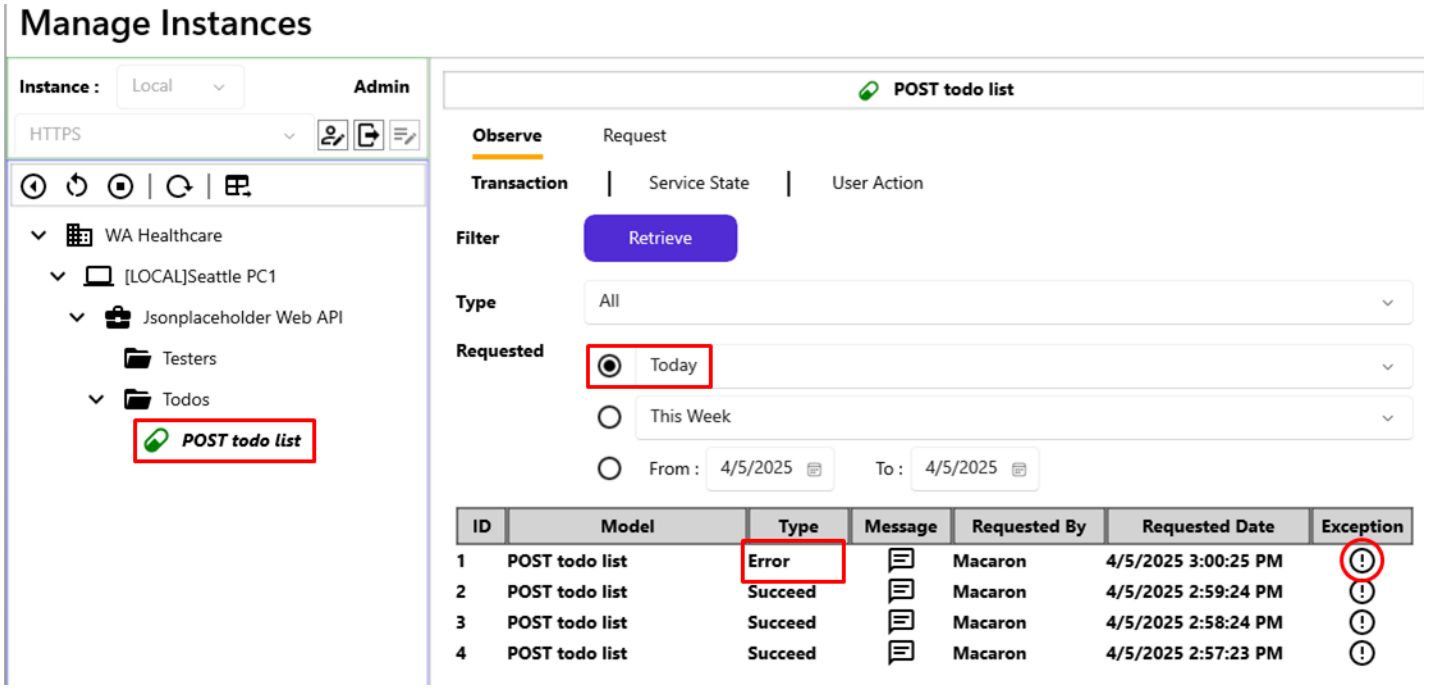


Figure 271.Transaction history of the POST todo list today

Previously, we kept the service running, and there was a new entry with an Error type at the top of the list after we deployed the modified group service. And that was thrown by the EnsureSuccessStatusCode() method after receiving a non-success status code due to the request with the invalid route value "files" instead of "todos." Let's see what exception data is in there. Click the first entry's **Exception** (ⓘ) button.

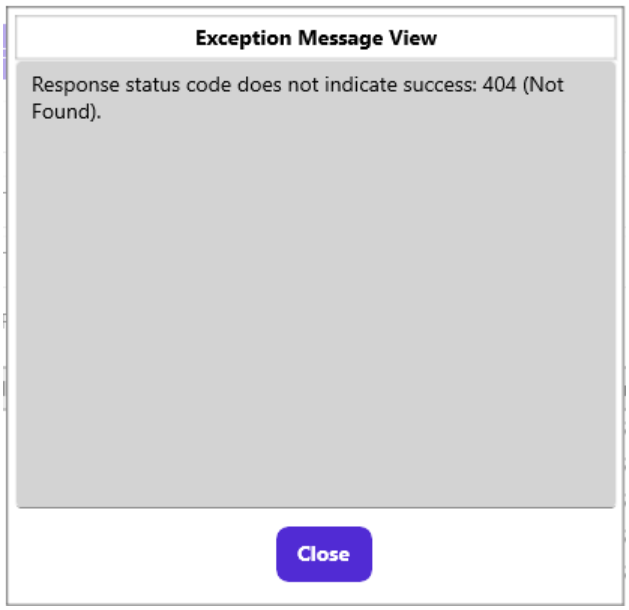


Figure 272.An exception message handled by Maca

The exception message thrown by the EnsureSuccessStatusCode() explains that the HTTP POST request was invalid, i.e., couldn't find the endpoint, and thus returned the 404 error, one of the non-success status codes.

Restore the route value and redeploy

We saw what could happen when we used the EnsureSuccessStatusCode() method in case of a non-success status code. So, we can restore the value with "todos" to see if the running service works well after the value restoration. Please go to the Development workstation, and select the Todos node on the project explorer. Then update the value "files" to "todos" and click the Save (💾)

button. As we did, **load only the group** to the server. Once the load is done, go back to the Management workstation and click the Source tab. Click the refresh ( ) button to make sure the latest group is loaded to the server. Select the Todos group and deploy the service. Likewise, **deploy the group only**.

Retrieve the history of the service

As we fixed the route value, running POST todo list service will point at the correct endpoint and get the success status code. Now, let's check what the transaction history looks like. Select the POST todo list node and click the "Retrieve" button under the Observe>Transaction tab, as shown in Figure 273.

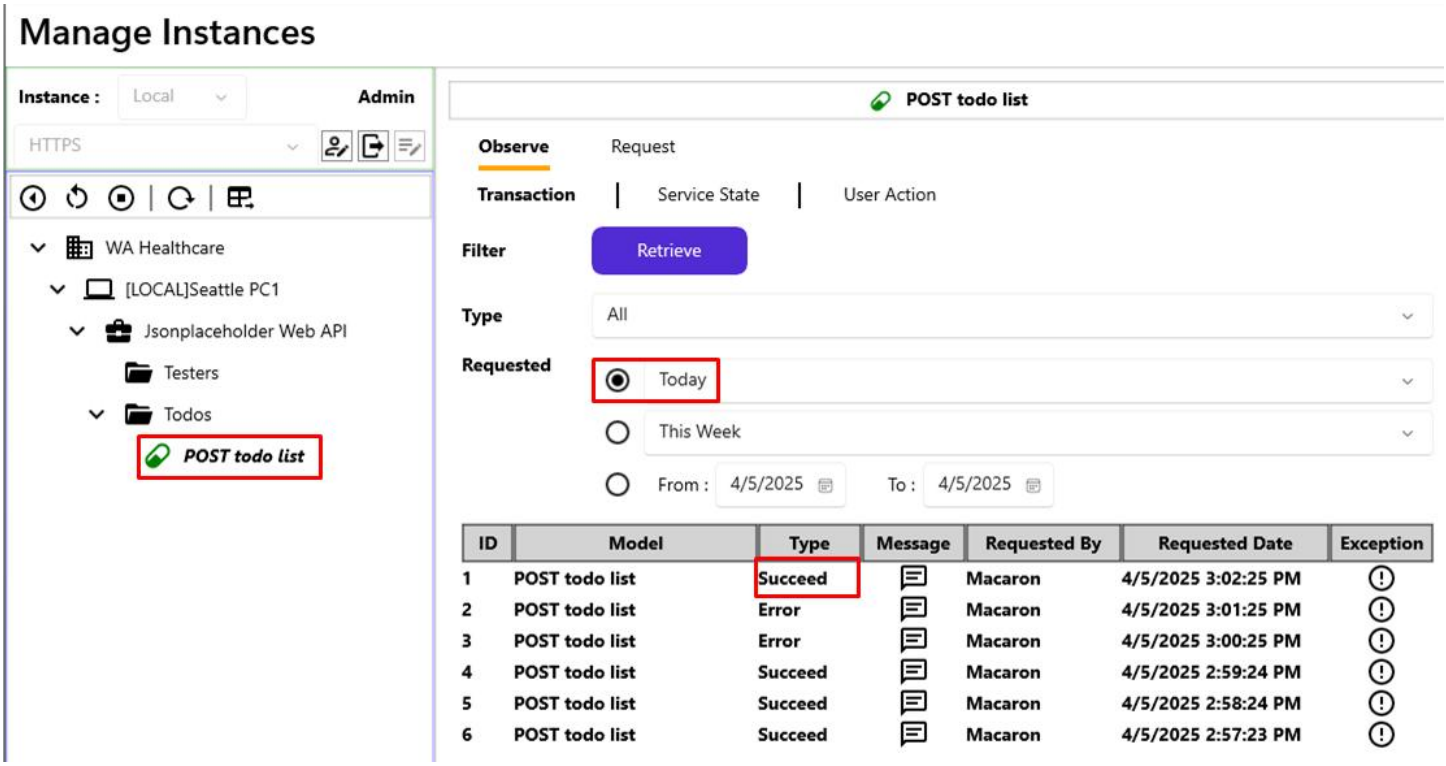


Figure 273.Transaction history of the POST todo lsit service

As you expected, the running service now works as normal, and the HTTP POST was processed successfully as a result. So we can see the text files are generated again after 2 minutes, as shown in Figure 274.

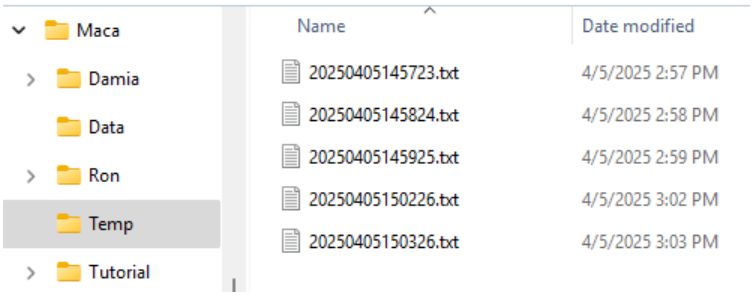


Figure 274.Generated text files

In the above example, we used the EnsureSuccessStatusCode() method to set all non-success status codes as Errors. You can keep this style if you don't have to handle each, or specific, non-success status code. But in many cases, you may need further error handling, such as call other services or do some more work based on the returned status codes. So, please write the error-handling code based on the requirements or given circumstances.

Execute the HTTP PUT

The PUT method is to update the existing data that we submitted to the server before. So, the server always checks the identifiable id that is assigned to the object when it was created on the server. A popular example of this would be a system-generated number that increases by 1. Once the object is found, i.e., there is existing data, the server will update the object based on the submitted object. Otherwise, the server will return a predefined non-success status code, such as 500 InternalServerError, for example. But if the submitted object is invalid, e.g., the object type is different, the required value or property is missing, or the value is invalid, the server will return the corresponding non-success status code as a result. Let's try these cases in the new service.

Under the same project, create a new service named "PUT todo list" with a CodeRepeator type. Once the service was created, go to the Execution > Steps view. Then click the Web API button to open the UI. As we did before, set the base address and route with the corresonding References. Withouth changing anything, just click the Execute button.

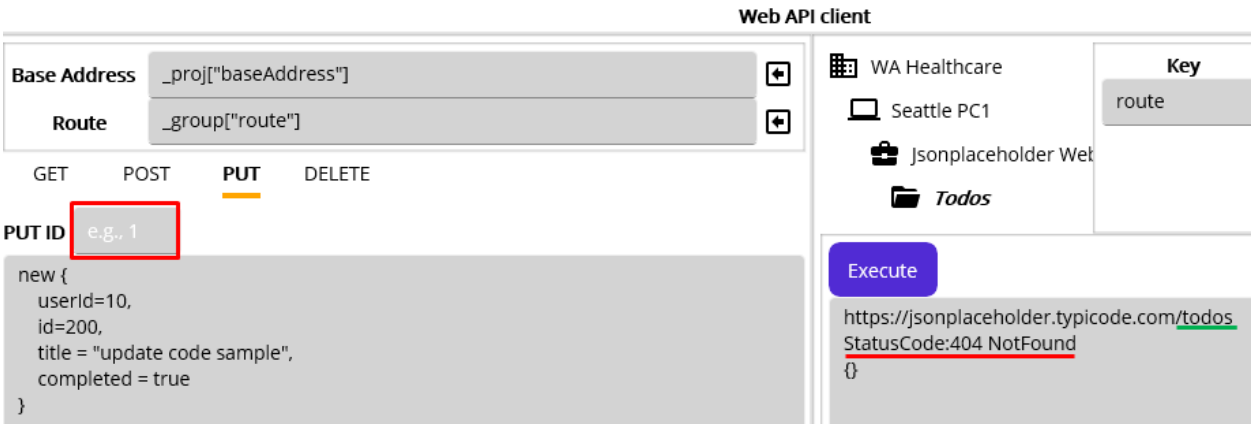


Figure 275.PUT request with no PUT ID

As we didn't provide the PUT ID, we received the 404 NotFound response, as you seen in Figure 275. If you look at the endpoint, the last part is "../todos" instead of "../todos/10" for example. As briefly mentioned above, the server checks the identifiable id. But there was no id value presented in the request and thus returned 404 as a result. With this example, it is clear that we have to provide an identifiable value for the HTTP POST.

Note. Like GET ID, the PUT ID here is what the server requires. Depending on the server's definition, it could be interpreted differently as well. For example, if the endpoint is for the products, "PUT ID" could mean "PUT PRODUCT ID."

Now, type 200 for "PUT ID" and execute again.

Figure 276.HTTP POST with id 200

As you saw in Figure 276, the server returned a success code 200, and we had no issue to submit the data. But at this time, we can identify the endpoint is different from previous one, i.e., "../todos/200." With this example, we understand that the submitted id 200 was identified in the server, i.e., id 200 was in the database. So, the server fetched the existing object and replaced it with the submitted object. And the updated value was returned as a result.

Now, let's try PUT with a different id. Please type 201 in the PUT ID area and click the Execute button.

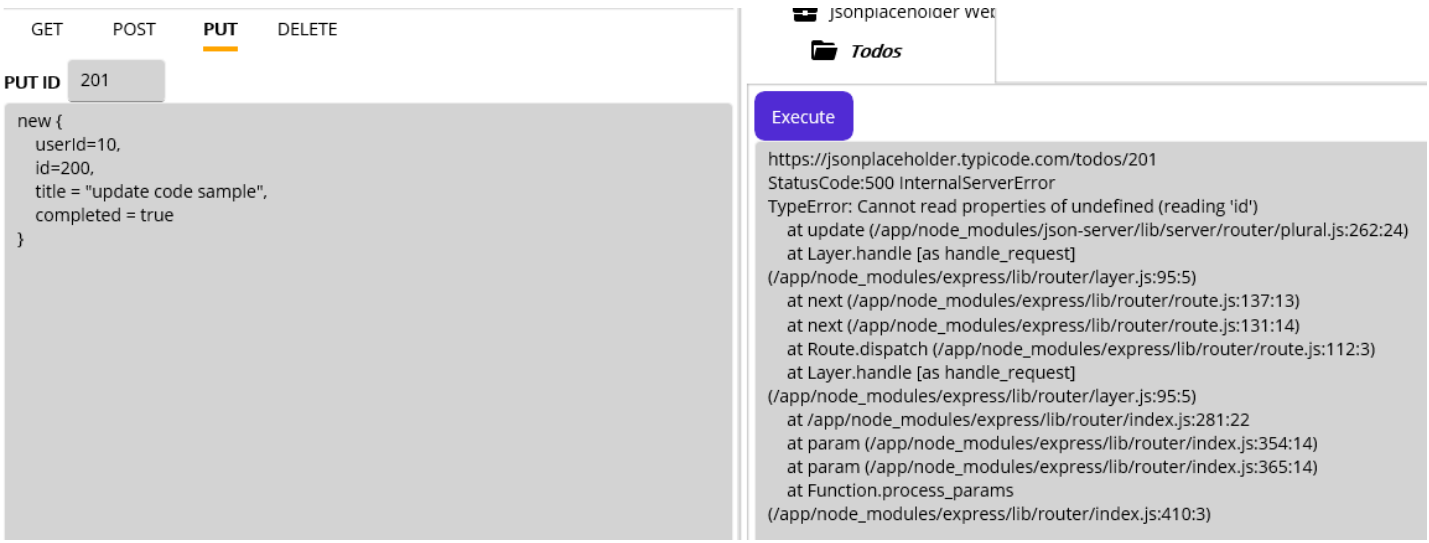


Figure 277.500 response with nonexisting id in the server

With the id 201, we received 500 status code at this time, as shown in Figure 277. Compared to the previous execution that used 200, we can assume that there is no object in the server with id 201.

Note. The Jsonplaceholder Web API site does maintain only 10 userIds from 1 to 10, and each userId has 20 ids. So, the last id in their database is 200. (You can check this by using GET with a different query string.) And we will get the expected results inside that boundary only. Even if we successfully submitted new id 201 with userId 11 in the POST section, there was no actual data creation in their server. And thus the PUT request with id 201, which is updating the data, was returned with the 500 status code. But as long as the id is identifiable, i.e., between 1 and 200, and all properties' values are provided, the PUT request will be successful.

So far, we have tested several cases for HTTP PUT method. Some cases will give you an idea of how to handle possible exception cases. You can execute more PUT requests within or outside the data boundary for your practice. Then, we can insert the code. Click the Insert code.

```
// Creates short-lived HttpClient instance
using HttpClient httpClient = _httpClientFactory.CreateClient();
httpClient.BaseAddress = new Uri(_proj["baseAddress"]);
StringBuilder routePutId = new();
// Route
routePutId.Append(_group["route"]);

// PUT ID: Replace the value with a variable, e.g., /{_id}
routePutId.Append($"/200");

//Please set corresponding HttpContent
var jsonPayload = new {
    userId=10,
    id=200,
    title = "update code sample",
    completed = true
};
using JsonContent httpContent = JsonContent.Create( jsonPayload );

using HttpResponseMessage response = await httpClient.PutAsync( routePutId.ToString(), httpContent );

//response.EnsureSuccessStatusCode();

using JsonDocument jsonDoc = JsonDocument.Parse( await response.Content.ReadAsStringAsync() );

// Use response.StatusCode and jsonDoc.RootElement
```

Figure 278.Inserted auto-generate PUT method code

As you see in the Figure 278, there would be no significanltly different code compare to the previous GET and POST examples. What you can check first is the PUT ID section to replace it with the variable or passed parameter to make sure the request is different in each execution. Then modify the jsonPayload's property values, i.e., replace the actual value with the variable or passed parameter. And you can put the data saving code that we used in the previous section.

Since you can make this sample code work exactly same as other services, i.e., dump the response in the text file, we will not cover the deployment and display the result. You can try that parts for your practice later. So, we can move on to the DELETE part, which will be quite similar in functioning.

Execute the HTTP DELETE

Create a new CodeRepeater service named "DELETE todo list" under the Todos group. Go to Execution>Steps view and click the Web API button. As we did, set the base address and route with the References.

Figure 279.HTTP DELETE method section

As you see in the Figure 279, we have the DELETE ID area to configure for the DELETE method. If you remember the PUT method that uses the id to find the object in the database, you will notice that the DELETE ID will be used to find the object in the database as well. So, the DELETE ID is what we have to provide in order to delete the associated object stored in the database.

As we know that the id ranges from 1 to 200, we can use one of them. Let's try id 200, which is the last one. Type 200 in the DELETE ID area and click the "Execute" button.

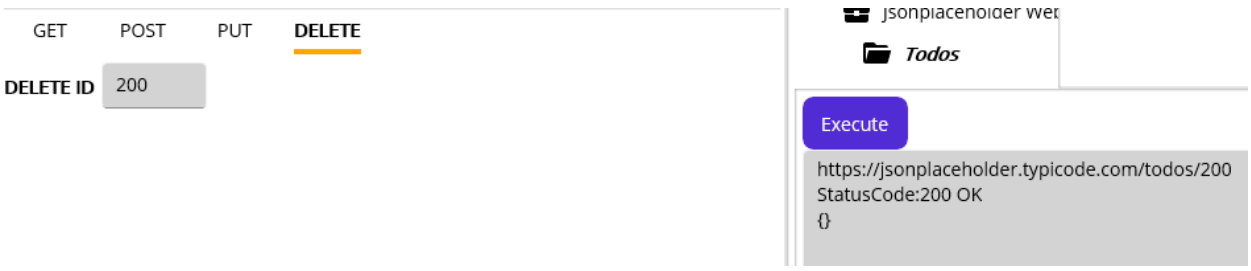


Figure 280.Sucessfully deleted item with id 200

As you see in the response, status code 200 was returned, i.e., the server deleted the object related to the id 200. But again, the server does not actulate manipulate the data, we just assume that the data was deleted. Likewise, if we do not provide the DELETE ID, the server will return nonsuccess code, as shown in Figure 281.

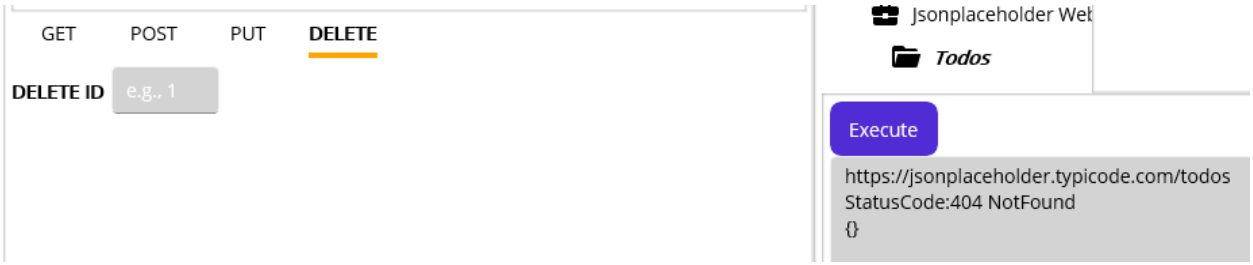


Figure 281.404 NotFound due to the missing id

This exception case is the same as what we saw in the PUT case, and we can skip the explanation.

Note. You can try exception case id 1000, which is not in their database, for DELETE and will see the unexpected success code, as shown below Figure 282. Although this would be a non-success case, the server returned 200 because they don't strictly check the id value and return 200 as long as the value is provided. So, please check the DELETE for your quick code review only.

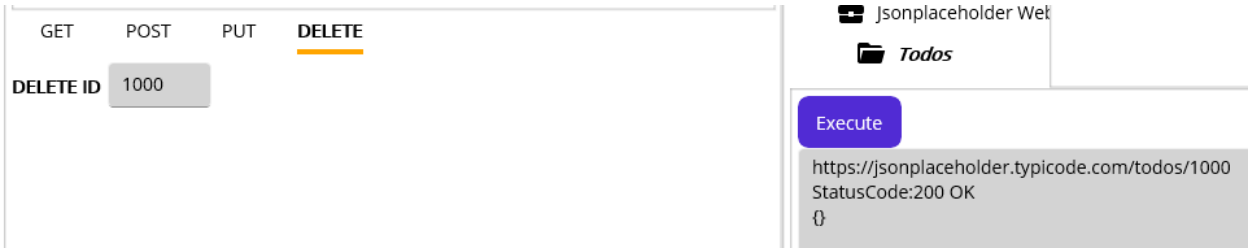


Figure 282.Successfully deleted item with id 1000

Since there is no specific topic to cover in the DELETE, let's insert the code. Click the "Insert code" button.

```
// Creates short-lived HttpClient instance
using HttpClient httpClient = _httpClientFactory.CreateClient();
httpClient.BaseAddress = new Uri(_proj["baseAddress"]);
StringBuilder routeDeleteId = new();
// Route
routeDeleteId.Append(_group["route"]);

// DELETE ID: Replace the value with a variable, e.g., /{_id}
routeDeleteId.Append($"/1000");

using HttpResponseMessage response = await httpClient.DeleteAsync( routeDeleteId.ToString() );

//response.EnsureSuccessStatusCode();

using JsonDocument jsonDoc = JsonDocument.Parse( await response.Content.ReadAsStringAsync() );

// Use response.StatusCode and jsonDoc.RootElement
```

Figure 283.Inserted auto-generated DELETE method code

As you noticed, this code is also quite similar to the previous one, and you can easily follow the code. So, we are not going to go deeper and move on to the next section.

Summary

As we created sample services followed by the HTTP methods, we could easily identify what each service actually does. Inside the searvice, we could test HTTP methods on the fly and insert corresponding auto-generated code to expedite service implementation. Although the service type used here was a CodeRepeater, we can set the service as RequestListener, which we will cover shortly, to support specific cases. And the project represented the target destination and could contain URI information that can be referenced in the underlying services. Lastly, the group may have the route value that also can be referenced in the containing services. By applying this structure, we will implement the actual requirements in the next section. But before that, we still need to test or practice more on file uploading because we just submitted a JsonContent that contains only the primitive types, such as string, int,

or boolean. As your first assignment is uploading the image file, which is supposed to be in jpeg format, it needs to be tested with the POST method unless you are already familiar with the file uploading with a web API. To support that situation, Maca provides a testable web API, the Maca Web API. Since we don't maintain that on the cloud, we published a Docker image downloadable from DockerHub as an alternative. But one thing you need to do beforehand is to install **Docker Desktop**[61] (https://docs.docker.com/desktop/setup/install/windows-install/). We are not going to cover the Docker Desktop installationin this tutorial, so please check the installation site before you move on to the next steps.

Maca Web API on DockerHub

Once you installed the Docker Desktop on your machine, you are now ready to pull the image from Dockerhub. For this tutorial, we will use CLI (Comand Line Interface), especially the Windows Command Prompt, instead of the Docker Desktop Windows UI. If you use Visual Studio Code[62] (VS Code), each command typed in the Terminal will work as well. So, please follw the given command in each step carefully.

Note. Before running docker commands, Docker Desktop must be started. And we assume that you already have a docker account.

First of all, please open Command Prompt, if necessary as Administrator. To pull the image, you can visit the Maca's DockerHub repository (https://hub.docker.com/r/bighillsoftware/maca-web-api) and copy the Docker Pull Command. Or you can copy the below command and paste it to the command propmt directly.

```
docker pull bighillsoftware/maca-web-api:v1.0
```

Once the download is complete, you can check the images using beow command.

```
docker images
```

Then you will see the result similar to following Figure 284.

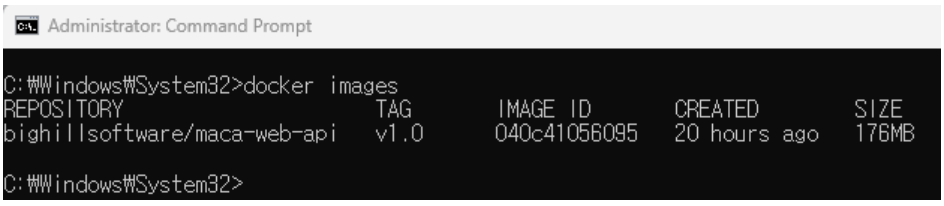


Figure 284.List of the docker images

Note. You can check the image list from VS Code or even on the Docker Desktop UI. All commands we will see can be done in those apps as well. So, please try them when you have time. For the details of commands used here, you can visit https://docs.docker.com/reference/cli/docker/image/ and check each command.

Run Maca Web API

Although we have a web api docker image, it is still not ready to run. Since the web API requires a certificate to run on your machine and accept HTTPS requests, we need to have a certificate and register it in on your machine. In our local testing case, we need a self-signed certificate. There are a couple of utilities to do the job. But we will use .NET SDK's dev-certs, which provides short steps.

.NET 9 SDK

Since Maca supports .NET 9, we assume you may already have .NET 9+ on your machine. If not, please install .NET 9 (https://dotnet.microsoft.com/en-us/download/dotnet/9.0) because we will use some utilities that are part of its SDK. For the detail about the installation on Windows, you can visit the site (https://learn.microsoft.com/en-us/dotnet/core/install/windows). Also you can visit the site (https://learn.microsoft.com/en-us/dotnet/core/tools/) to learn more about the .NET CLI related to the "dotnet" command.

[61] https://www.docker.com/products/docker-desktop/
[62] https://code.visualstudio.com/

dotnet dev-certs

As we mentioned, you need a certificate, i.e., self-signed certificate, to run the web API on you machine. The dotnet dev-certs will create a self-signed certificate and make it trusted by your machine. One minor change from .NET 8 to .NET 9 is that the utility doesn't create a folder to save a generated certificate. So, you have to create a folder first and specify it on the command. For this tutorial, we will use the below folder for the storage:

C:\Maca\Cert

You can create the folder with a different name in the other location where you prefer, but please make sure the location is intact because it will be referenced when the web API runs. Now, you can run below command to create a certificate and save it to the created folder.

```
dotnet dev-certs https -ep C:\Maca\Cert\MacaWebApi.pfx -p macapa55
```

As we created the folder, "**-ep**," which is an export path, is composed of the folder and certificate file name, i.e., **MacaWebApi.pfx**. Then "**-p**," which is a password, is set as "**macapa55**." But you can set it with what you prefer.

Note. For your reference, you can visit the site (https://learn.microsoft.com/en-us/aspnet/core/security/docker-https?view=aspnetcore-9.0) to learn about hosting Docker with HTTPS. And you will notice that the above command is similar to the site's example. For the complete details about the dotnet dev-certs command, like clean or import, please visit https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-dev-certs.

Once you run the command, you will see the generated certificate file, as shown in Figure 285.

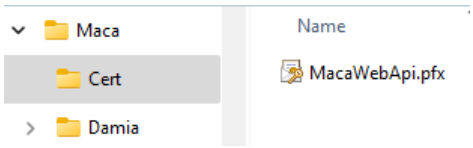


Figure 285.Generated self-signed certificate

Now, we will register this certificate and make our machine trust it using dev-certs. Copy below and paste It to the command prompt then run it. And the certificate will be successfully trusted by your machine.

```
dotnet dev-certs https --trust
```

Note. We may run the web API in HTTP mode only for the local testing purpose. But in most cases, you will need an HTTPS connection, as we did in the Jsonplaceholder example above. So, it will be a good practice to register and trust localhost certificate.

So far, we created a self-signed certificate and make our machine trusted it. The next step is making the web API to use that certificate when it starts.

Run Docker image

Now, we are almost ready to run the image. You can copy below command and paste it to the command prompt.

```
docker run --rm -p 5000:80 -p 5001:443 -e ASPNETCORE_URLS="https://+;http://+" -e ASPNETCORE_HTTPS_PORTS=5001 -e ASPNETCORE_ENVIRONMENT=Development -e ASPNETCORE_Kestrel__Certificates__Default__Password="macapa55" -e ASPNETCORE_Kestrel__Certificates__Default__Path=/cert/MacaWebApi.pfx -v C:\Maca\cert:/cert/ bighillsoftware/maca-web-api:v1.0
```

At first look, it is quite an overwhelming command, but Table 30 will describe the detail of the above command.

Options	Values	Details
--rm		Clean up(remove) when the container(web API) stops

-p	5000:80 5001:443	Port mappings Local-port:Container-port
-e	ASPNETCORE_URLS="https://+;http://+"	Supports http & https
-e	ASPNETCORE_HTTPS_PORTS=5001	Sets which port is for HTTPS
-e	ASPNETCORE_ENVIRONMENT=Development	Development or Production
-e	ASPNETCORE_Kestrel__Certificates__Default__Password="macapa55"	The password used in the self-signed certificate
-e	ASPNETCORE_Kestrel__Certificates__Default__Path=/cert/MacaWebApi.pfx	The location of the self-signed certificate. Combined with the volume
-v	C:\Maca\cert:/cert/	Physical location info mapped to the volume
	bighillsoftware/maca-web-api:v1.0	Image name:Tag 

Table 30.Docker run command details

You can keep using the command. But if you want to modify, i.e., use different folder or port, you can change the value with preferred value. For example, if you want to use different ports, you can replace 5000 and 5001 with something else like 8000 and 8001. Of course, ASPNETCORE_HTTPS_PORTS=5001 should be changed as well. In addition, if you stored the certificate in a different location, you can replace the value in -v option with a corresponding value. For the complete options of docker run, please visit https://docs.docker.com/reference/cli/docker/container/run/.

Once you paste the above command and press enter key, you will see the Maca Web API is running, as shown in Figure 286.

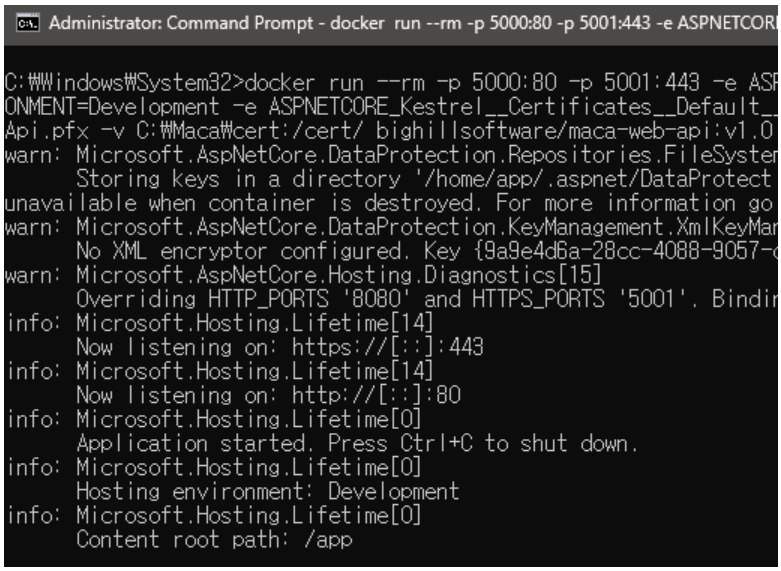


Figure 286.Running Maca Web API

One thing to note is the ports in Figure 286 are the container's ports. So, pelase meke sure to use local machine, i.e., your machine's port that you defined in the docker run comand. So, the port that we will use is 5001, which is HTTPS.

Maca Web API documentation

Before we consume the services, it is good practice to know what the API provides. Unlike typical web API that uses Swagger[63] for the API documentation and testing, Maca Web API provides OpenAPI documentaion that can be displayed using "https://localhost:5001/openapi/v1.json," as shown in Figure 287.

[63] https://swagger.io/

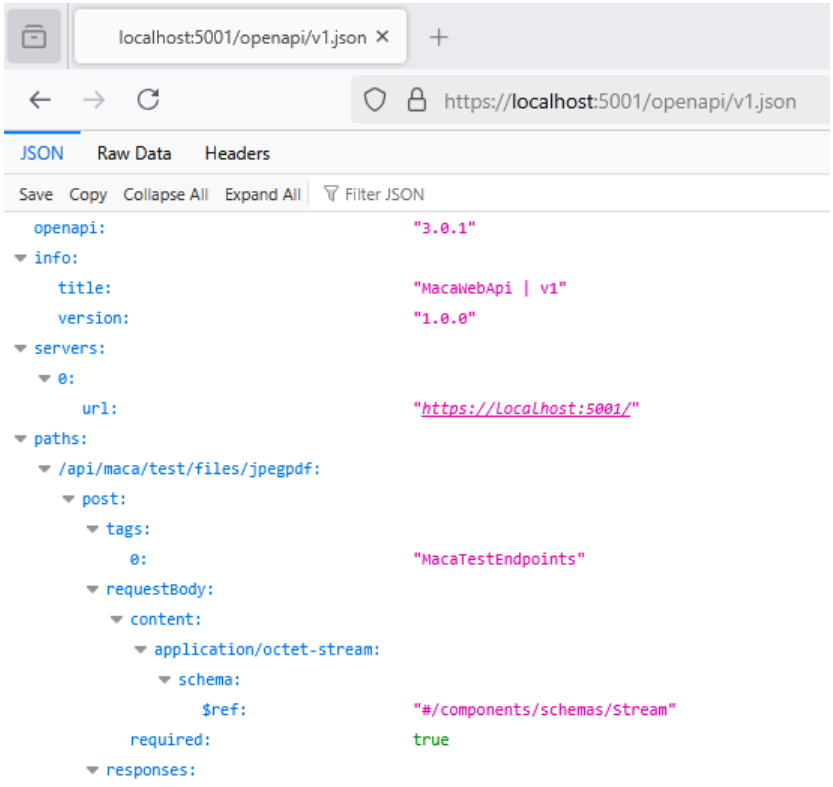
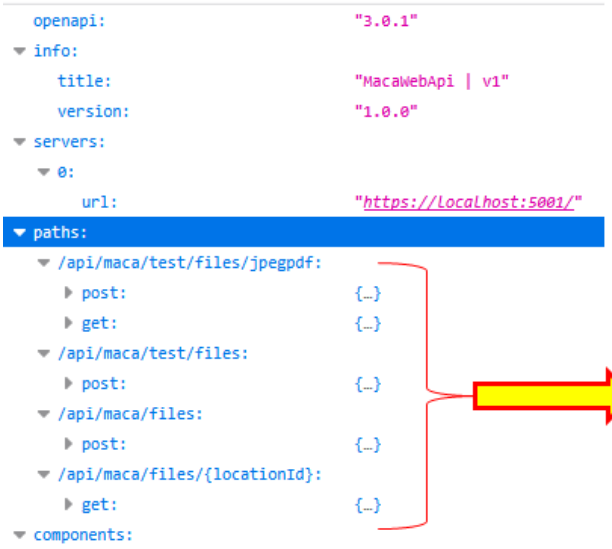


Figure 287.Maca Web API with openapi spec

If you are familiar with Swagger, you will be a little bit frustrated with the interface because it doesn't provide a way to consume the API on a web browser. The main purpose of the Maca Web API is to support Macadamia, specifically the Web API client UI that we used in the previous sections. All the method in the API are designed to be consumed in the Web API client UI or in the deployed MacaronService. As we tried the Jsonplaceholder web API, we can now test the Maca Web API in the same way. But before that, let's take a quick look at the spec, as shown in Figure 288.



Types	Values	Examples
Base Address	https://localhost:5001	
Route	api/maca/test/files/jpegpdf	
	POST	PostAsync
	GET	GetAsync
Route	api/maca/test/files	
	POST	PostAsync
Route	api/maca/files	
	POST	PostAsync
Route	api/maca/files/{locationId}	
	GET	routeQuery.Append($"/1"); GetAsync

Figure 288.Maca Web API specification

As you see both sides in Figure 288, you will easily notice that we have 4 routes and each has callable method(s). If you look at the right side table, what will be consumed is more clear. If you remember what we defined in the Web API client UI, there are two fields in the target destnation, i.e., Base Address and Routes. Followed by the openapi spec, we can use the values to populate those

fields. For example, the first route “api/maca/test/files/jpegpdf” has two methods, POST and GET. Now let’s see what’s in the first route, as shown in Figure 289.

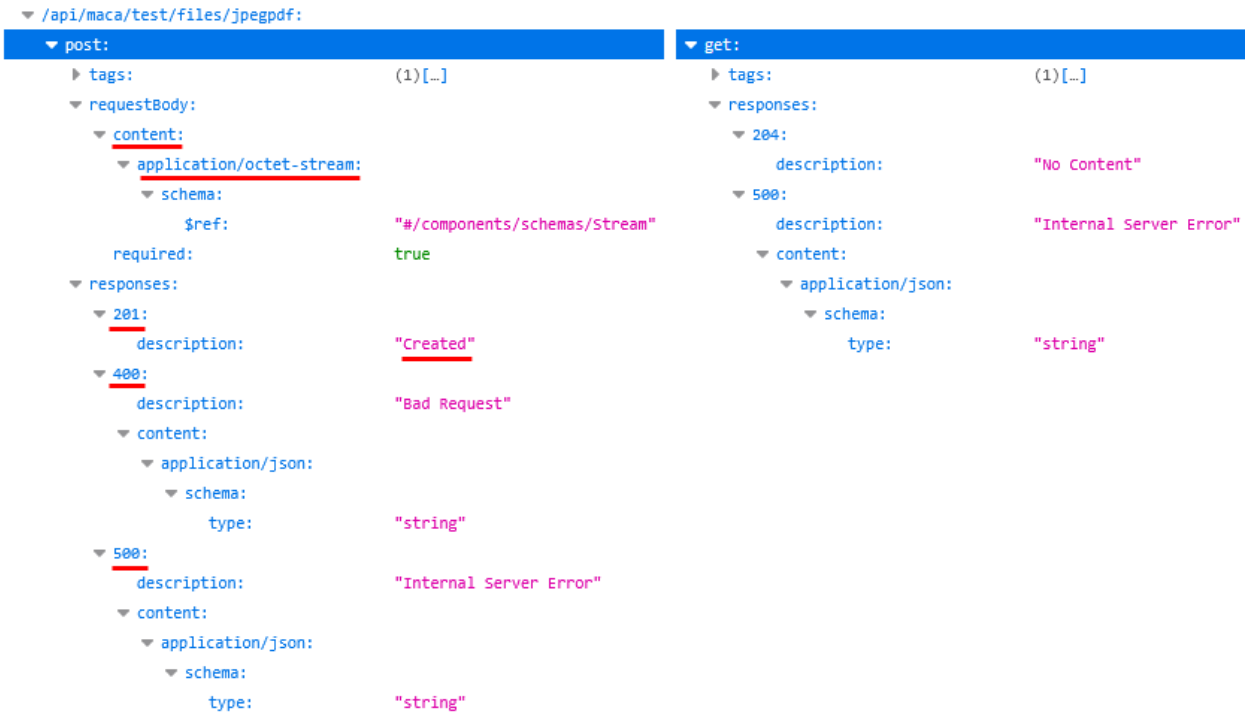


Figure 289.jpegpdf route detail

As you see, the post requires “octet-stream,” which is StreamContent or ByteArrayContent based on the user coding approach. And the “Created” message will be returned when the server successfully processed the request, i.e., 201 case. Then you will notice what will be returned in case of 400 and 500 cases. And the get, which is not fully documented for the 200 case, will return the file as a Stream. The spec will be modified in the next release for more annotations, but this will give you an enough information about the service comsumption. Using this spec, let’s test the first route with POST and GET.

Note. The above Figures are captured from the FireFox browser. If you use other browsers, such as Chrome, you may see plain text format json data. To see the similar view, you can download the json file and open it in the json viewer software. Or use the online JSON viewer.

Testing image upload and downlod with Maca Web API

Before we test the API, let’s create a project Maca Web API as we did for the Jsonplaceholder API like Figure 290.

Figure 290.Maca Web API project with baseAddres reference

Followed by Figure 288, we can add **baseAddress** with "**https://localhost:5001**" and let that be referenced by the containing services. Then we can create the group followed by the route name. For now, we are not going to implement actual services. As we just practice the file handling services, we can keep using "Testers" folder to set the first route **fileJpegPdf** with "**api/maca/test/files/jpegpdf,**" as shown in Figure 291. "fileJpegPdfRoute" would be clearer to identify the key, but we used "fileJpegPdf" for shorter name. Please set the key as you prefer in your project. And don't forget to save both group models.

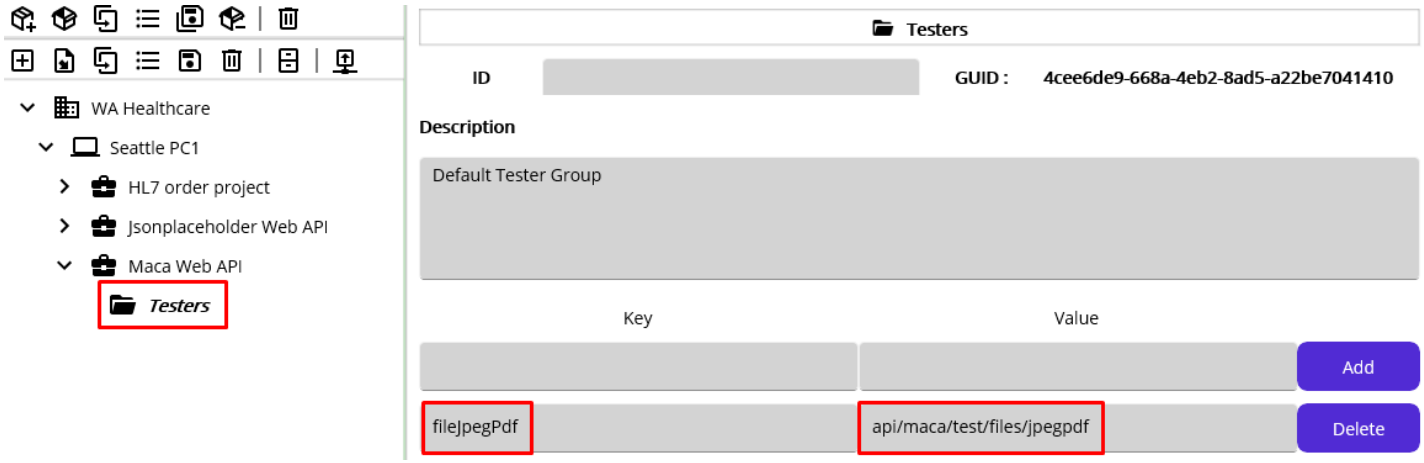


Figure 291.Route reference in the Testers group

Now, we are ready to use these two references in the new test services under the Maca Web API project. So, let's create a service that uploads a file to the Maca Web API. As you may notice, the first route's POST accepts either JPEG or PDF files. Any other file formats, such as txt or bmp, will not be accepted. So, our first test will be uploading those files. But before we upload, let's check which HttpContent we can use because what we used before was JsonContent, which is not proper for file uploading. Among HttpContents, what we can use in common for simple file uploading will be StreamContent and ByteArrayContent. The word "simple" here means upload file only. In case we need to upload complex data set including file, we can choose MultipartFormDataContent that we will cover in the later section. So, let's create a service that uploads a file using StreamContent.

Uplod a PDF file with StreamContent

As you remember, if the service consumes the Web API, we had set the service name to start with the http method, such as POST. And we are going to upload the PDF file first, so add PDF as well. Lastly, add StreamContent because we test StreamContent. With this name, create a CodeRepeater service for simple processing, as shown in Figure 292

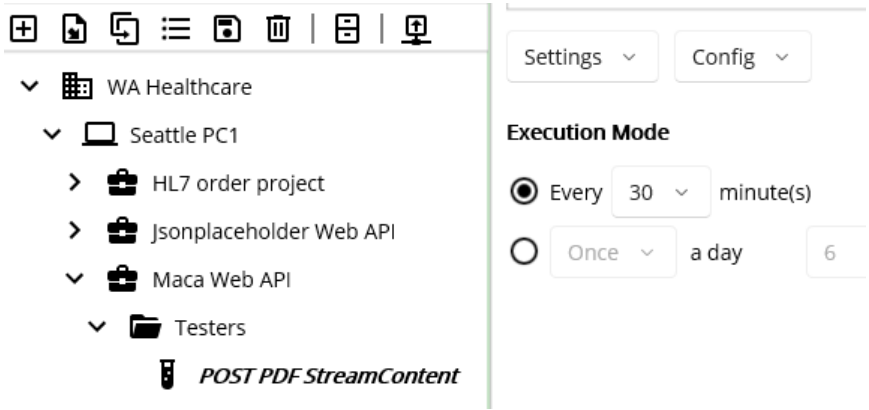


Figure 292.POST PDF StreamContent service

To avoid short-term processing that may cause too many transactions, set a 30-min interval. Then go to Execution > Steps and click the WebAPI (API) button to open Web API client UI. As we did before, browse the model explorer and set Base Address and Route with Project and Group model references, as shown in Figure 293.

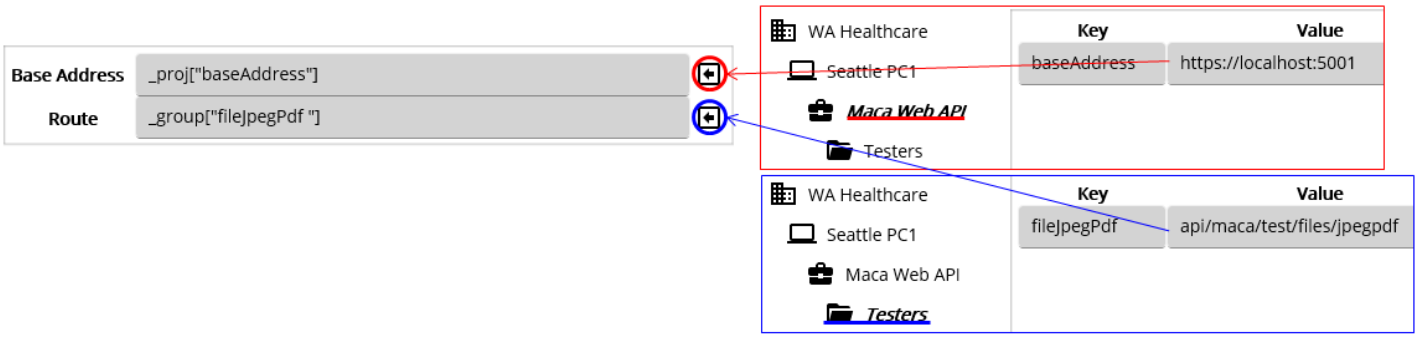


Figure 293.Set Base Address and Route with References

Previously, we used JsonContent for POST. As we are going to use StreamContent, please click the HttpContent picker and select StreamContent, as shown in Figure 294.

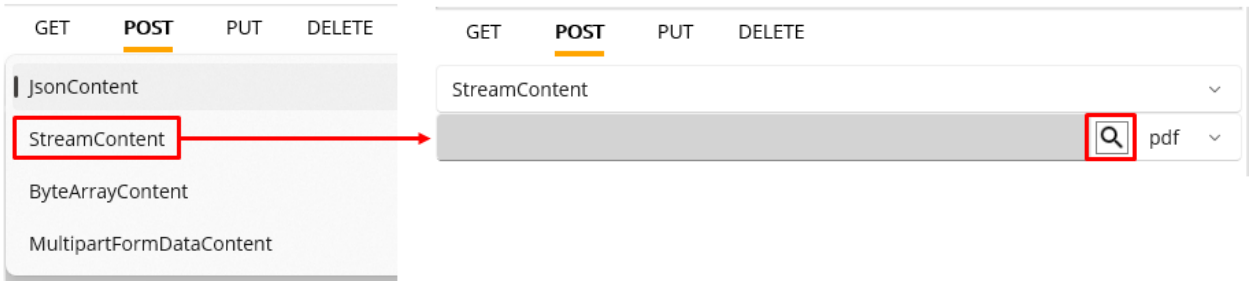


Figure 294.Select StreamContent for POST

As you see, there is a magnifier icon, which locates a PDF file. Click that button to select a PDF file on you machine. Once you pick the PDF file, click the Execute button. The final look will look like Figure

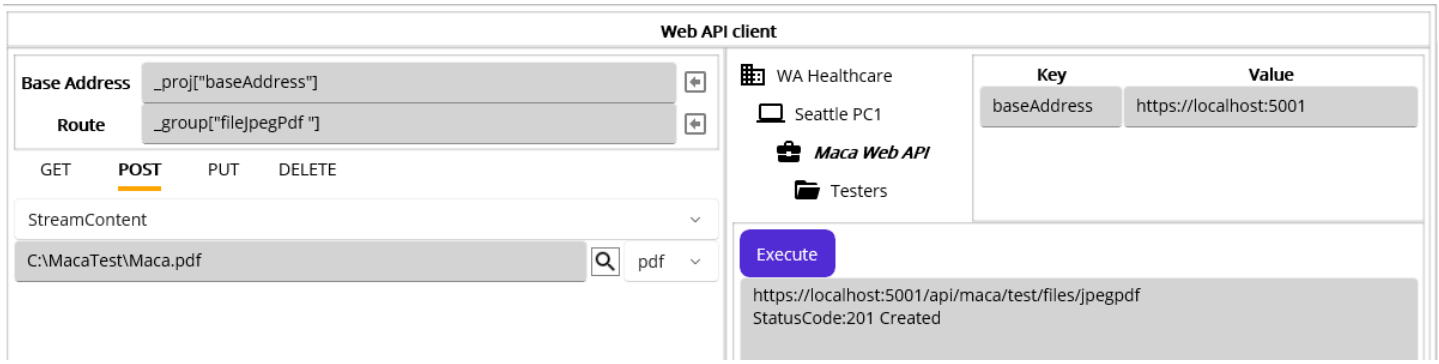


Figure 295.PDF file upload success case

If you look at the server, i.e., running container, there was a 1 message at the bottom, as shown in Figure 295.

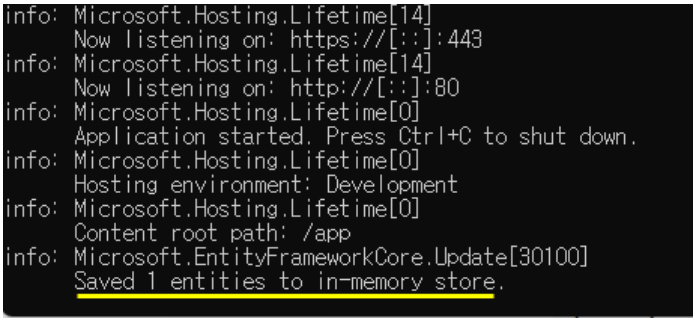


Figure 296.Upload file on the Maca Web API container

In both client and server sides, we saw the PDF file upload was successful. So, let's use this configuration and insert the code to the Code editor. Click "Insert code and close" button.

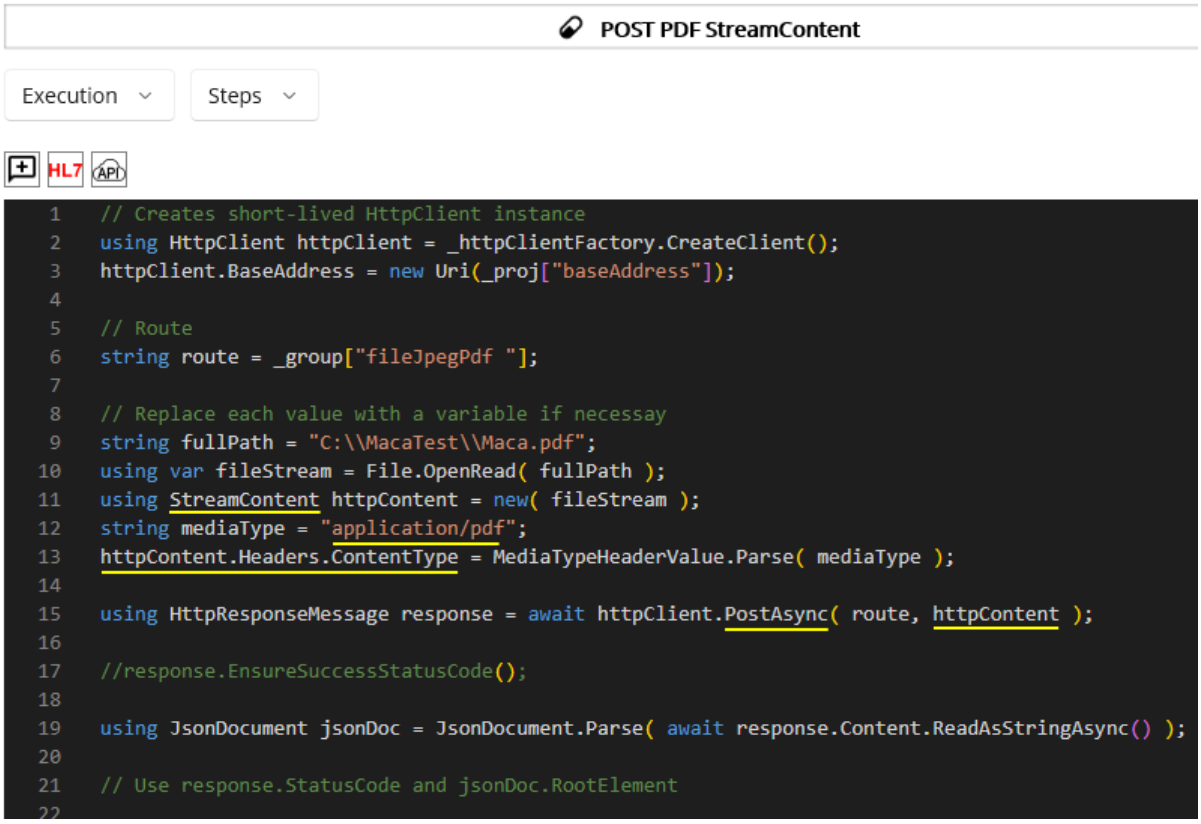


Figure 297.Inserted POST StreamContent code

As you notice, many parts of the inserted code share the JsonContent part. So, the key points here that you can take a look at are the StreamContent and the media type. Firstly, we located the file and read a stream from it using File.OpenRead. Then we created the StreamContent with the stream and set its media type as “application/pdf.” Lastly, set the StreamContent in the 2nd argument of the PostAsync.

Once we receive the response from the server, we may can the result message in the RonResult to save and see in the transaction. So, we can modify the code like below Figure 298.

```
using HttpResponseMessage response = await httpClient.PostAsync( route, httpContent );

result.ResultMessage += response.StatusCode + Environment.NewLine;
result.ResultMessage += await response.Content.ReadAsStringAsync();
```

Figure 298.Set response message to RonResult

For now, we are not going to load the service. When we finish the other services, POST ByteArrayContent and GET File, we will load and test them altogether. So, just save and verity the service at this point.

Upload a JPEG file with ByteArrayContent

As we tested PDF, we can test JPEG with ByteArrayContent at this time. Since the process to create the service will be exactly the same as the POST PDF StreamContent, you can follow the previous steps to create POST JPEG ByteArrayContent service.

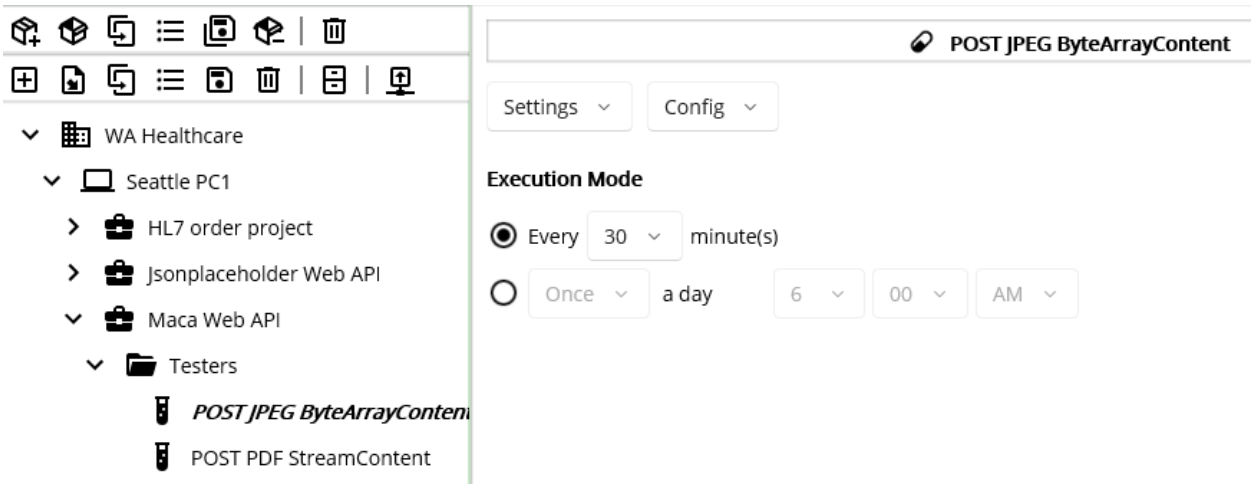


Figure 299.POST JPEG ByteArrayContent service

Once you create the service. Go to Execution > Steps and click API ( ) button to open Web API client UI. If you follow the previous steps, the final look of the POST will be, as shown in Figure 300.

Figure 300.POST JEPG ByteArrayContent service

Please note that we selected ByteArrayContent from the HttpContent picker and selected jpeg for the file type picker at this time. Now, click the magnifier button to locate the jpeg file. Then click the Execute button, as shown in Figure 301.

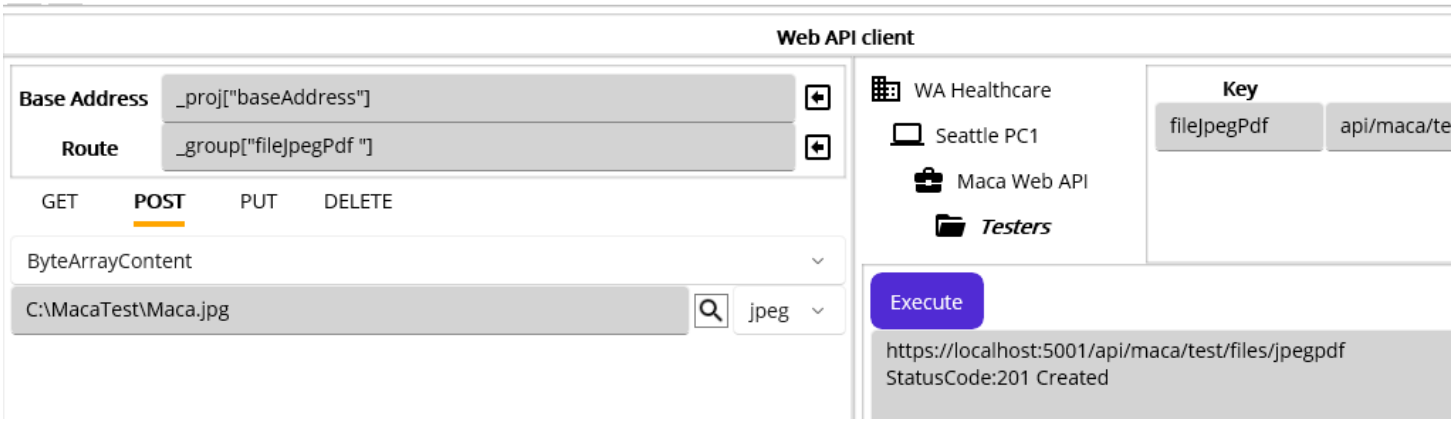


Figure 301.JPEG file upload success case

From the UI perspective, there is no huge difference between StreamContent and ByteArrayContent. So, we can check the difference in the inserted code in the code editor. As we saw the upload was successful, click the “Insert code and close” button.

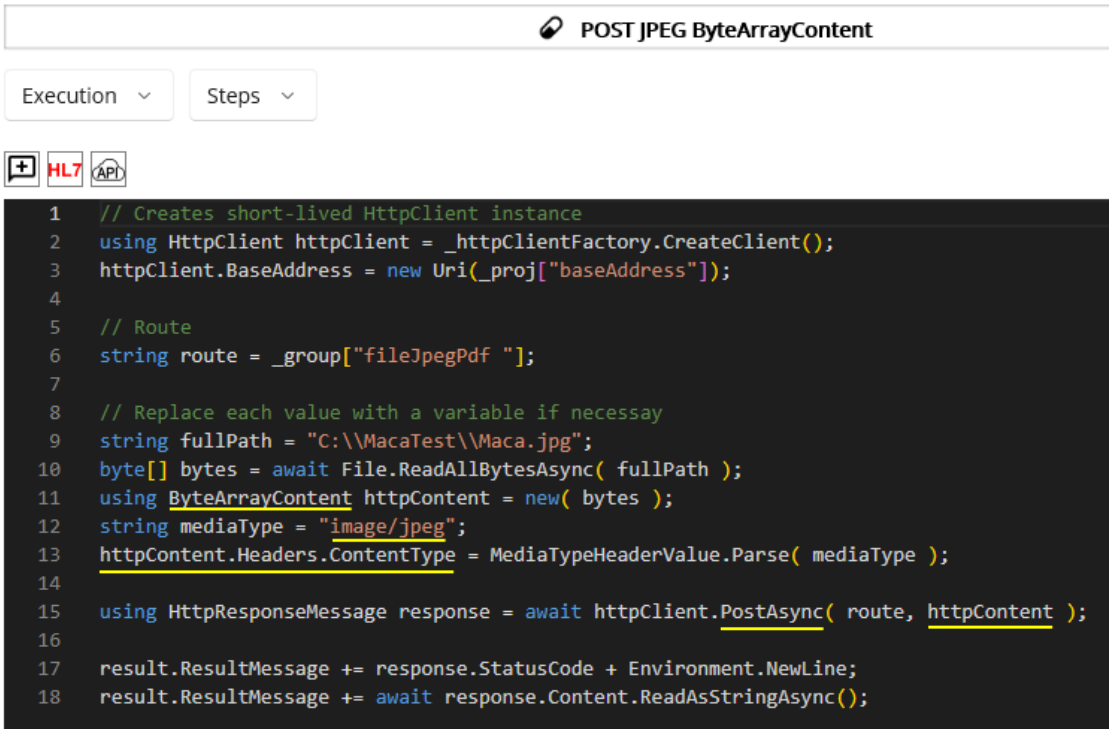


Figure 302.Inserted and modified POST ByteArrayContent code

As you see in Figure 302, the process of ByteArrayContent is quite similar to the one of StreamContent. We read the byte array using File.ReadAllByteAsync. Then we created ByteArrayContent with the byte array and set its media type as "image/jpeg." And the content is used in the 2nd argument of the PostAsync. Lastly, we add the result message along with the status code in the RonResult to save and see in the transaction.

So far, we uploaded 1 PDF file and 1 JPEG file to the Maca Web API. Followed by Figure 288 and 289, there is another GET method that downloads the image file. Although the description is limited, the server will return one file at a time. When the next request comes, the server checks next available file. If there is no file to return, it will return 204 "No Content." As you may notice, we have to save the download file, which comes as a stream. If we use the Web API client UI, we can't save the file because there is no option to save the file. Still, we will see 200 status code but the data in the response area will look awkward. So, let's move on the GET service to see what described here.

Note. Maca Web API uses an in-memory database, which means the data and files are not persistent. When you shut down the server, all the files along with the data you submitted will be removed.

Download a file

If you look at Figure 289, you will notice that there is no ID or query string associated with the GET method. So, we can assume that this GET will return an array of the files similar to Figure 243. Although the description is limited, the server will return 1 file at a time. So we have to handle 1 file download and save it as either PDF or JPEG. Based on this, create the GET JPEG or PDF file service, which is similar to the previous services.

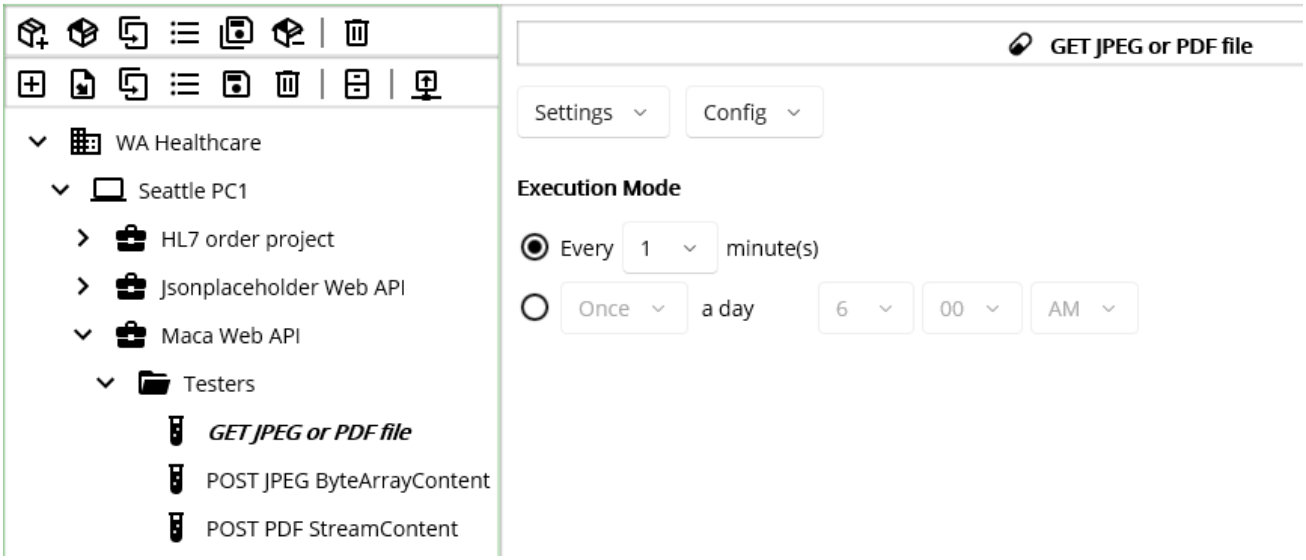


Figure 303.GET JPEG or PDF file

Once you create the service, go to Execution > Steps. Then click API (API) button to open Web API client UI. Set the target destination and keep GET tab, as shown in Figure 304.

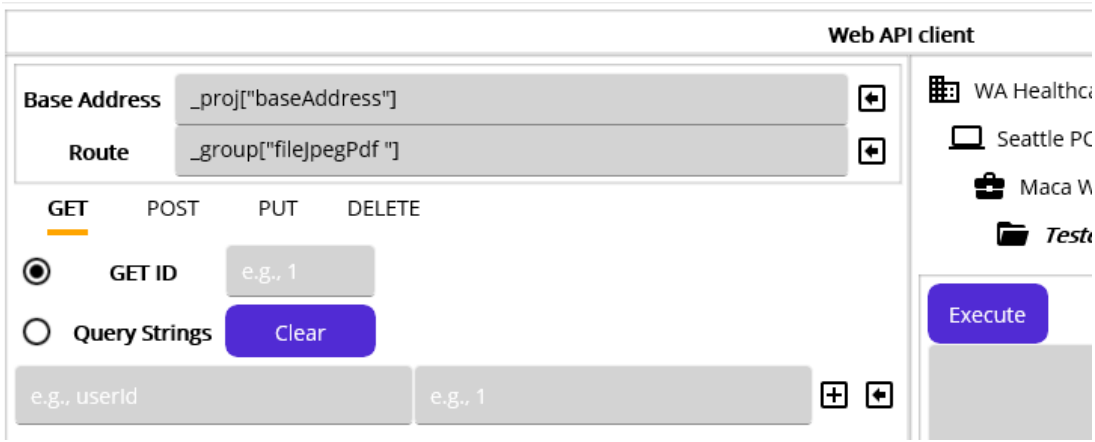


Figure 304.GET JPEG or PDF file Web API client

Before you click the Execute button, please check again the process of the GET that returns 1 file at a time. If you click the Execute button 2 times, all uploaded files will be marked as downloaded on the server. And the next time when you click the Execute button, no file will be downloaded. Now, let's click the Execute button.

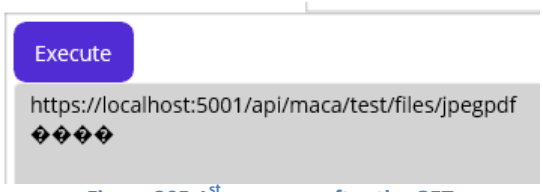


Figure 305.1st response after the GET

As we mentioned, there is no download related code in the UI and resonse area displays broken data as a result. But we didn't have any error from the server, so click the Execute button again.

Figure 306.2nd response after the GET

As we expected, we sill have awkward code in the response area. But we know the wer requested two times, and there will be no files to return on the server. So, click the Execute button again, which is the 3rd time execution.

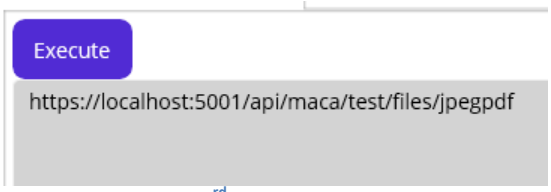


Figure 307.3rd response after the GET

At this time, we have an empty response, which is different from the previous executions. Based on this we can assume that there was not file download on the server. Although Figure 289 shows that the 204 “No Content” will be returned in case of no data to download, the response area shows notthing because it is not designed to handle that. So, we have to handle that situation in the code editor after insert the code. Click “Insert code and close” button.

Note. Since the Web API client UI’s Execute button doesn’t support the file download, which requires download data parsing, conversion, and download location setup, all above executions are not for actual testing. All testing should be done in the deployed services that handle file saving from the downloaded stream.

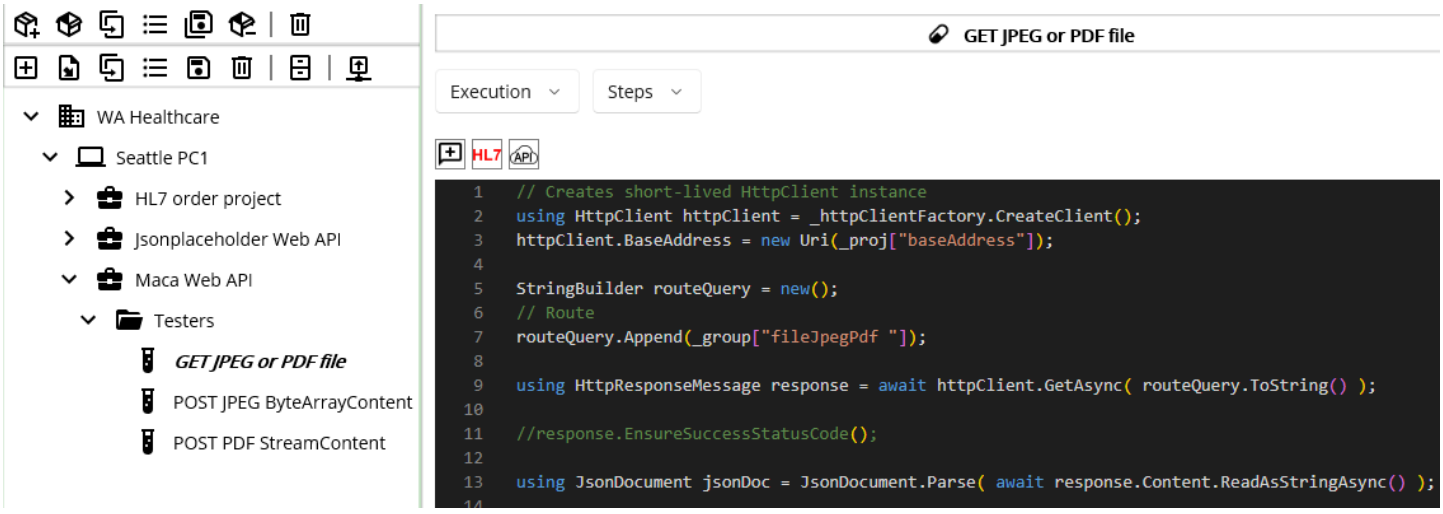


Figure 308.Inserted auto-generated GET code

As you see Figure 308, there is no code that handles response that returns stream data. So, we need to add file handling code, as shown in Figure 309.

```
response.EnsureSuccessStatusCode();
```

```
// status code 204 NoContent
if( (int)response.StatusCode == 204 )
{
  result.ResultMessage = "No file to download";
  return result;
}

using Stream stream = await response.Content.ReadAsStreamAsync();

string fileDir = $"C:\\Maca\\Temp\\Files";
if (!Directory.Exists(fileDir))
    Directory.CreateDirectory(fileDir);

string fileName = response.Content.Headers.ContentDisposition.FileNameStar;
string filePath = Path.Combine(fileDir, fileName);

using var fileStream = new FileStream(filePath, FileMode.Create, FileAccess.Write);
await stream.CopyToAsync(fileStream);

result.ResultMessage = fileName + " successfully download";
```

Figure 309.File download code example

The first thing we use differently is uncommented “resonse.EnsureSuccessStatusCod();” If you look at Figure 289, you will notice that there is 204 and 500. In success case, the server will return file. If there is no file, then it will return 204. Otherwise, the server will return 500. So, we can use EnsureSuccessStatusCode to catch all non-20x cases. And we will process only the 20x case below. For 204 case, we will set RonResult with the custom message “No file to download” instead of the default “No Content” message to avoid the confusion. Then we get Content’s data as a Stream. As you see, the file name(containing file extension) can be extracted from the Content’s Header value and we used that to define the complete file location. Finally, we saved the stream to the defined folder using CopyToAsync. As an extra step, we set ResultMessage with the file name to be saved and see in the transactions.

Note.The above code is handling what Maca Web API returns. You need to consult your client to ask what the API actually returns in which file type. Then build a code accordingly.

Run the services to upload and download files

So far, we created 3 test services that upload and download files. Please see the below table that compares the services.

Services	Current code
POST PDF StreamContent	Upload pre-defined PDF file every 30 min
POST JPEG ByteArrayContent	Upload pre-defined JPEG file every 30 min
GET JPEG or PDF file	Download a file every minute

Table 31.JpegPdf route related services

When we deploy all 3 services, 1 PDF file and 1 JPEG file will be uploaded immediately. Since the GET JPEG or PDF file runs every minute, it will download 1 file from the server. After 3 minutes, there will be no file to download. But if we keep running, after 30 minutes, the same files will be uploaded and the GET servcie will download again. So, let’s deploy them to see what will actually happen. But before deploy, please make sure verify all 3 services. If GET service complains with StringBuilder, then add “using System.Text;” in the Manage Service Libraries ( ).

Among them, deploy GET JPEG or PDF file fist. Once it is deployed, retrieve the transaction, as shown in Figure 310.

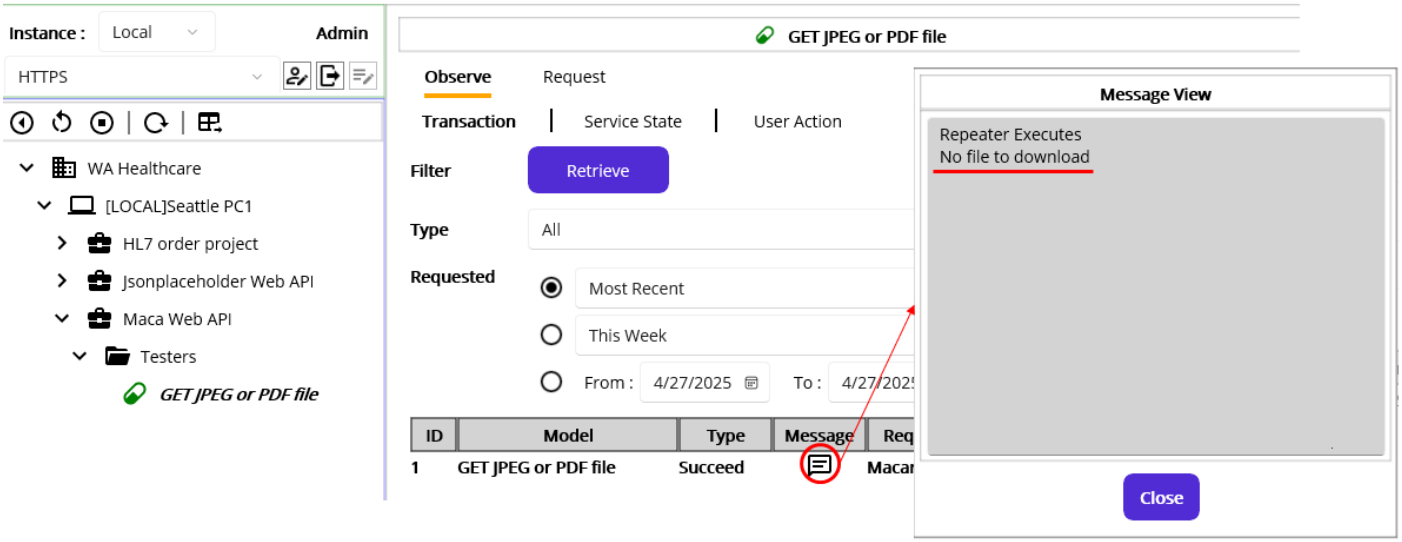


Figure 310.GET JPEG or PDF file transaction

As we clicked Execute button on the Web API client UI 2 times after POST two files, there is no file to download and Message View popup shows “No file to download” that we defined in the code. Let this service keep running. And before deploying the rest 2 services, please check the folder C:\Maca\Temp\Files exists because the service will create a folder when download the file. Now, deploy the rest 2 services. The following screen captures will show what you will see after deploying the services.

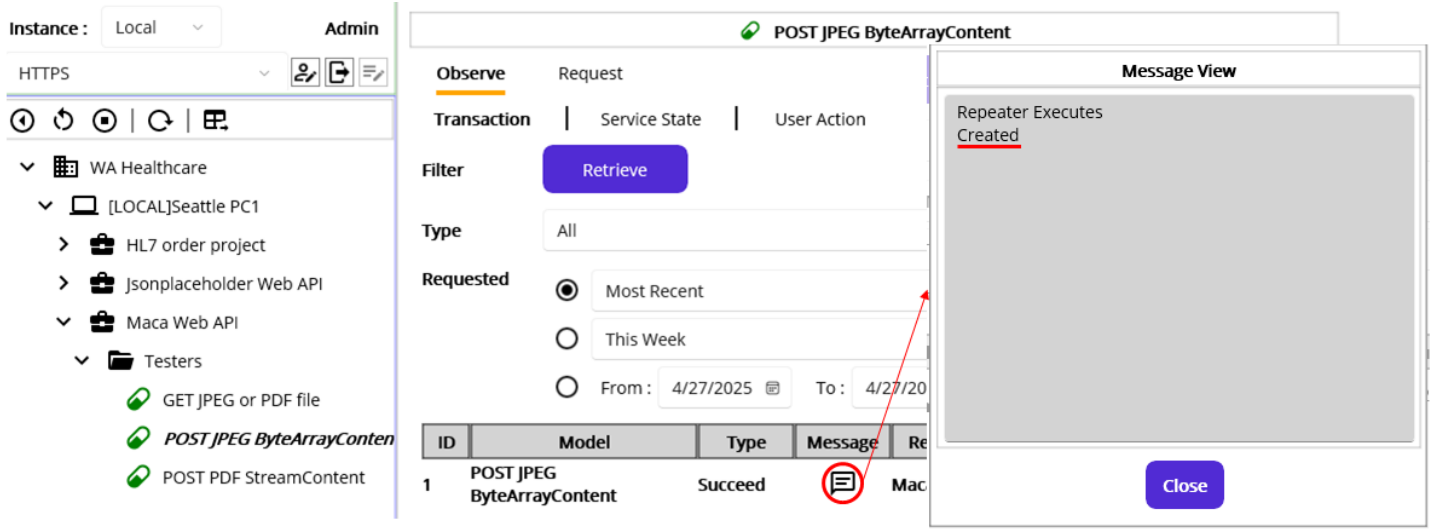


Figure 311.Uploaded JPEG file

Figure 311 shows the retrieved transaction of POST JPEG ByteArrayContent service and its ResultMessage containing StatusCode, Created.

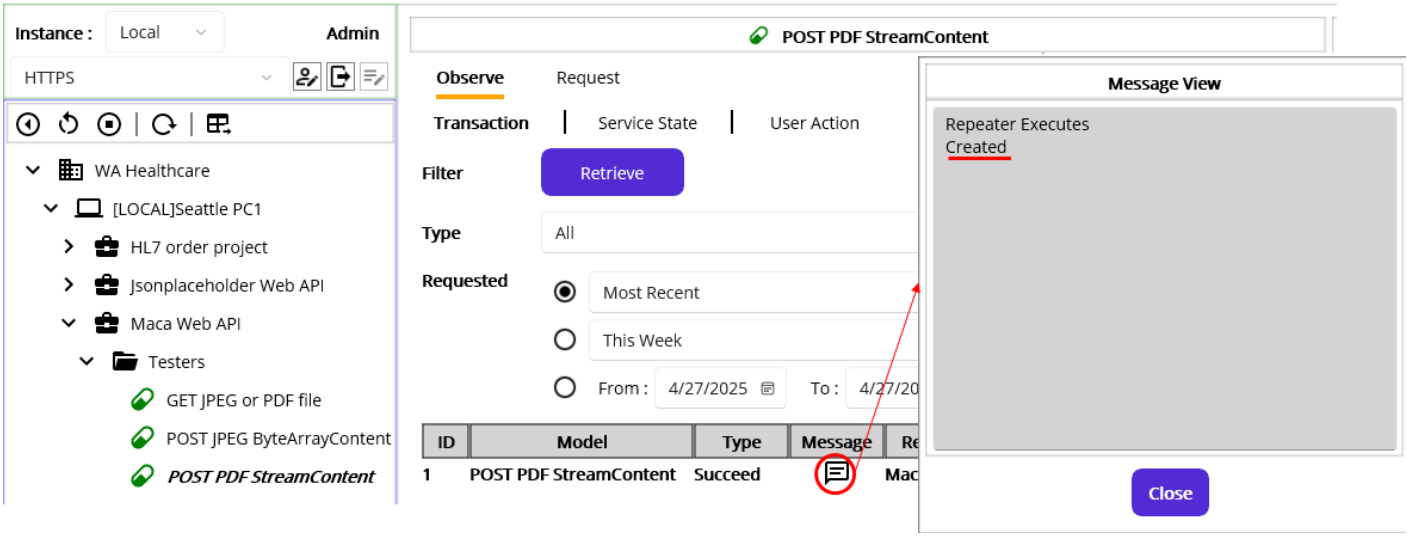


Figure 312.Uploaded PDF file

Figure 312 shows the retrieved transaction of POST PDF StreamContent service and its ResultMessage containing StatusCode, Created.

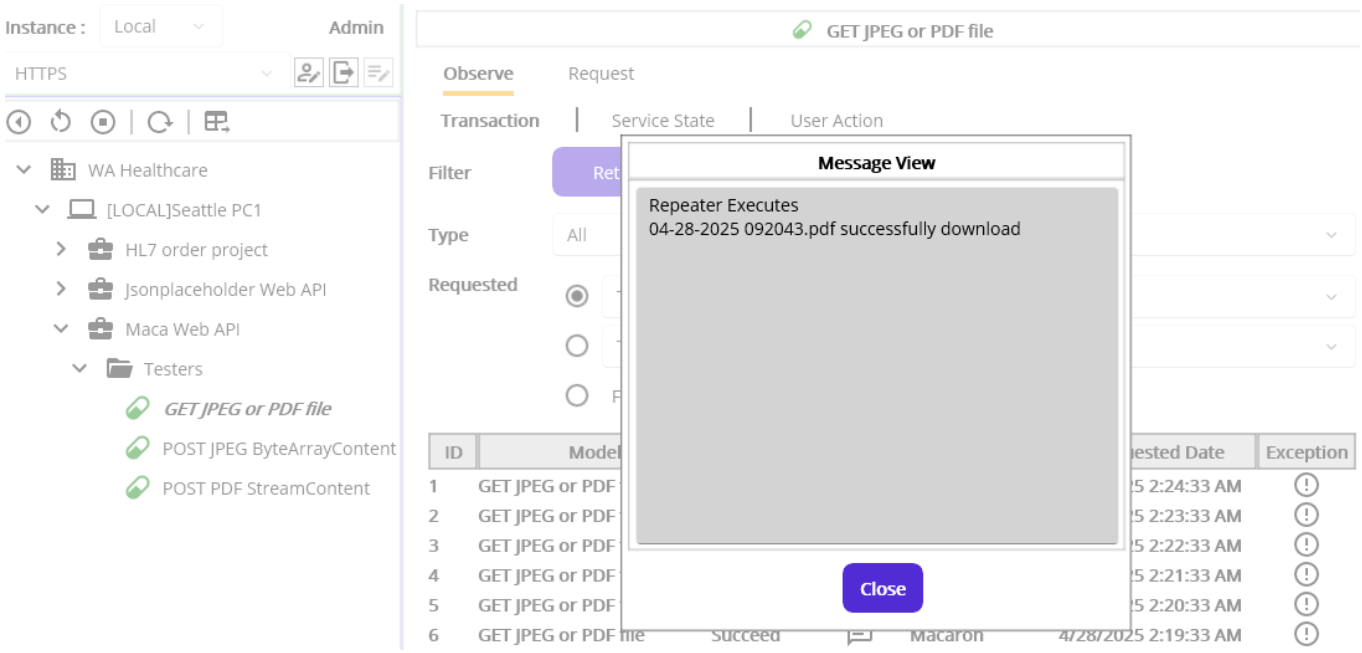


Figure 313.GET JPEG or PDF file transactions and its message

Figure 311 shows that the GET JPEG or PDF file service successfully download 1 pdf file. In the other transaction, you will see the jpeg file dowload message on the Message View popup. Since the service executes the code every minute, there are many transaction records in the result view. So, there will be many “No file to download” transactions, and you will see only two records that have downloaded messages. Lastly, check the folder C:\Maca\Temp\Files to see if there are any downloaded files. And you will see the two files, as shown in figure 314.

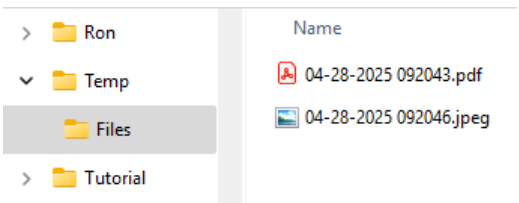


Figure 314.Downloade two files from Maca Web API

We will not test any exception case to move forward fast, so please try more test cases of your own by modifying the inserted code.

Complex data set

Previously, we created a service that uses only 1 HttpContent type. For examle, we used JsonContent for Jsonplaceholder's each HTTP method service. And we used StreamContent for POST PDF StreamContent and ByteArrayContent for POST JPEG ByteArrayConten for Maca Web API. In certain cases where the server requires a complex data set, those HttpContents can't be used due to their limited capabilities. Still, there is a way to submit some data in the Headers, we can't guarantee the values, not to mention the placeholders are limited and need an extra value conversion. In that case, the MultipartFormDataContent can be used.

Note. JsonContent sometimes can be used to submit and receive complex data set, e.g., embed file as Base64, it is more common to use MultipartFormDataContant, which is more flexible. In this tutorial, we will not touch that topic because that's out of the scope.

Upload a file with MultipartFormDatacontent

The MultipartFormDataContest can hold as many HttpContent as possible, and the type can be any of the StringContent, JsonContent, StreamContent, ByteArrayContent, and so on. In this tutorial, we will not compose complex MultipartFormDataContent. With 2 or 3 types of HttpContent, we will create MultipartFormDataContent. Although that will not be complex enough, it will give you a good hint to create more complex MultipartFormDataContent. To support this scenario, Maca Web API provides a service that requres MultipartFormDataContent. And the POST method belongs to the route "api/maca/test/files" is as shown in Figure 315.

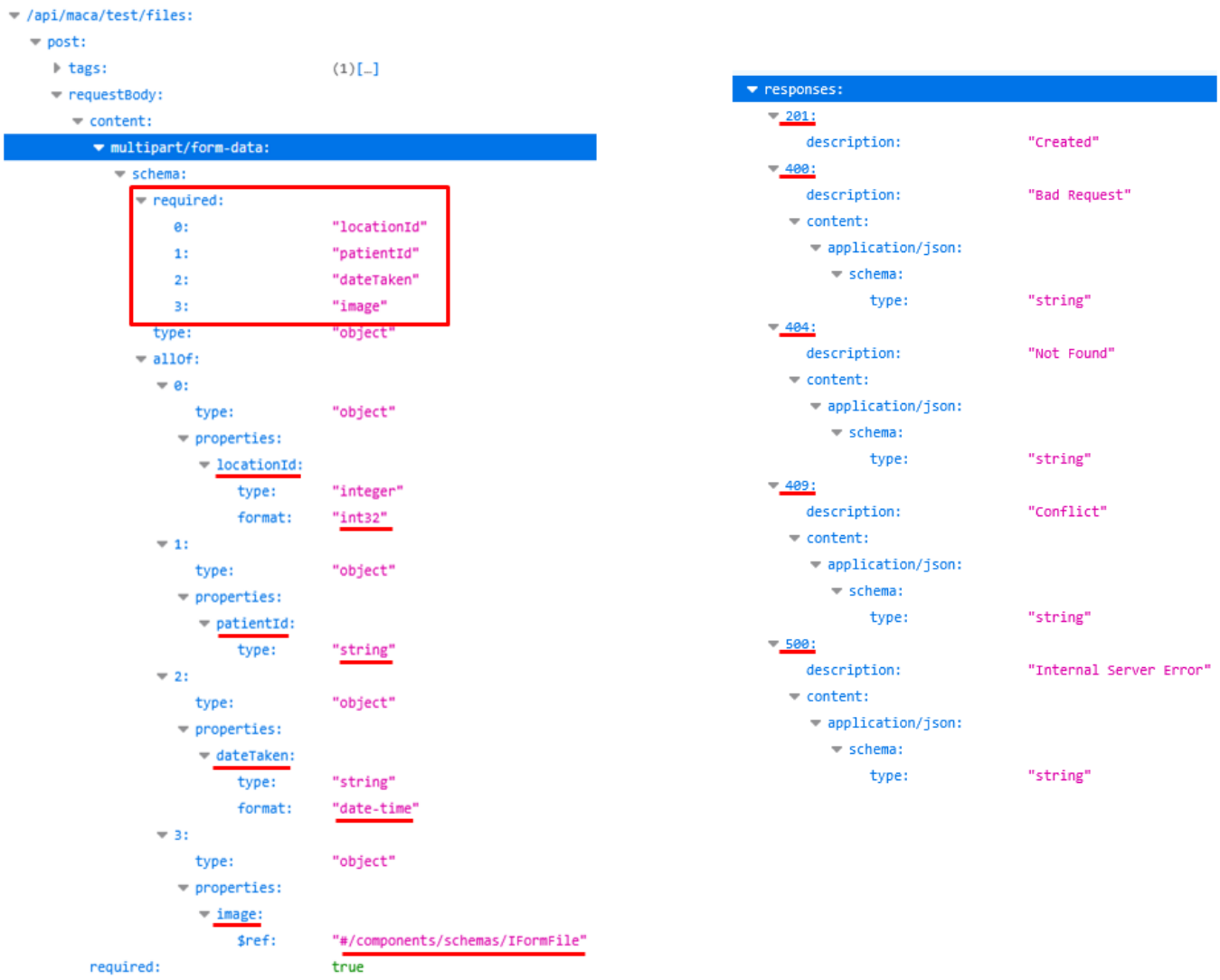


Figure 315.POST MultipartFormDataContent request(Left) and response(Right)

As you see on the right side, there are 4 required fields in the requestBody of the POST method. Each of the fields is described below in the form of indexed property. For example, the 1st property, which is index 0, is "locationId" with an integer type. The key point here is the property name "locationId." The API server strictly checks the name and will return an error if the name is incorrect or not provided. Then it checks the value is the expected type. Likewise, if the value is not the type or empty, the server will return an error as well. So, it is quite important to use right property name and value. And the 4th property is a file which is interpreted as IFormFile[64] on the server. So, we need to upload a file with property name "image." As we practiced in the above tests, we can use StreamContent or ByteArrayContent for this property value.

As you may notice based on the 4th property, **each property must be submitted as an HttpContent type**. For example, the 1st property is an integer type, but there is no HttpContent related to a numeric value. In this case, we can use StringContent to hold a numeric value. And of course, the 2nd property is StringContent type. Lastly, the 3rd property is also StringContent type, but the value format must be date-time, e.g., "2025-04-10T14:19:14." Since the HTTP request and response use the JSON format, which follows the ISO 8601[65], the date format will be a little different from the perspective of regular DateTime[66]. So, we have to populate the value in that format. You can check more about the examples of date format conversion in this site https://learn.microsoft.com/en-us/dotnet/standard/datetime/system-text-json-support.

And the right side of the figure shows what will be returned in case of an exception on the server side. As we are not going to explain them one by one, please check them by testing each error case by yourself.

Note. Based on the above description, you may think that the values could be populated in the JsonContent and submit that. However, if you look at the highlighted line on the left side of Figure 315, you will notice that the content is "multipart/form-data," which is MultipartFormDataContent. So, we must submit the MultipartFormDataContent when we call POST in the route "api/maca/test/files."

Like the previous services, let's create a service named "POST image MultipartFormDataContent." You may set the name differently, like "POST file Form" or "POST image Form," which describes the service better.

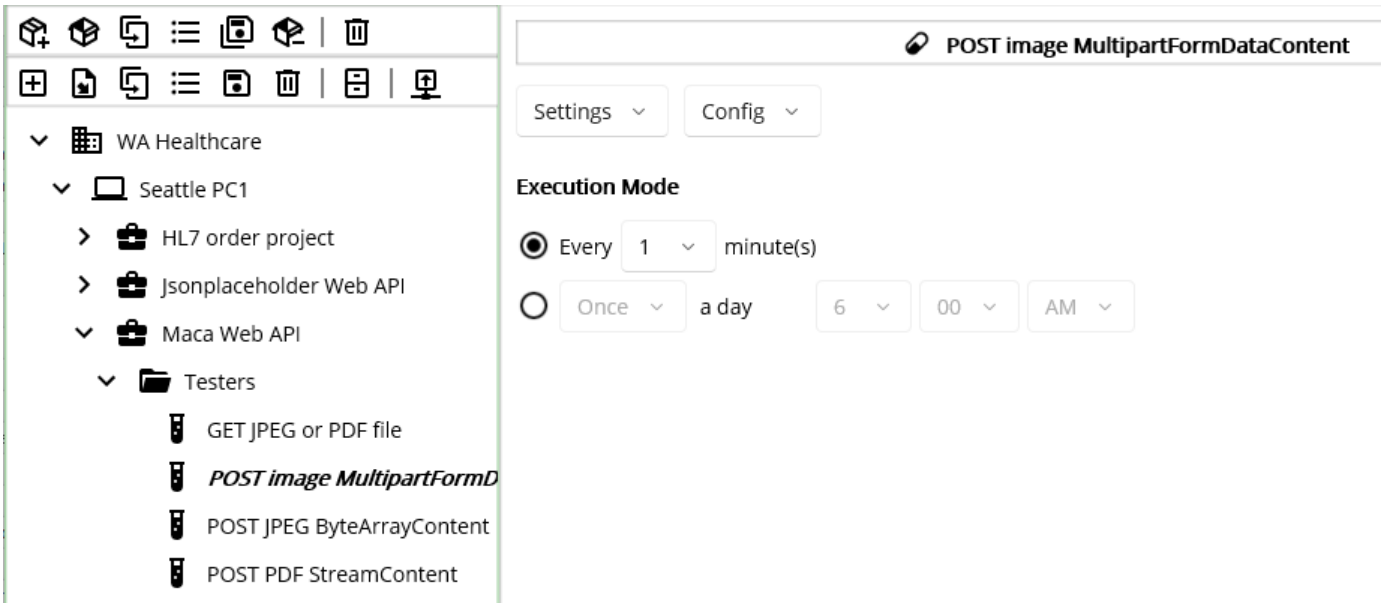


Figure 316.Created POST image MultipartFormDataContent service

Since we are working on the different route, i.e., api/maca/test/files, we need to add that in the group's References, as shown in Figure 317.

[64] https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.iformfile?view=aspnetcore-9.0
[65] https://www.iso.org/standard/70907.html https://www.iso.org/standard/70908.html
[66] https://learn.microsoft.com/en-us/dotnet/api/system.datetime?view=net-9.0

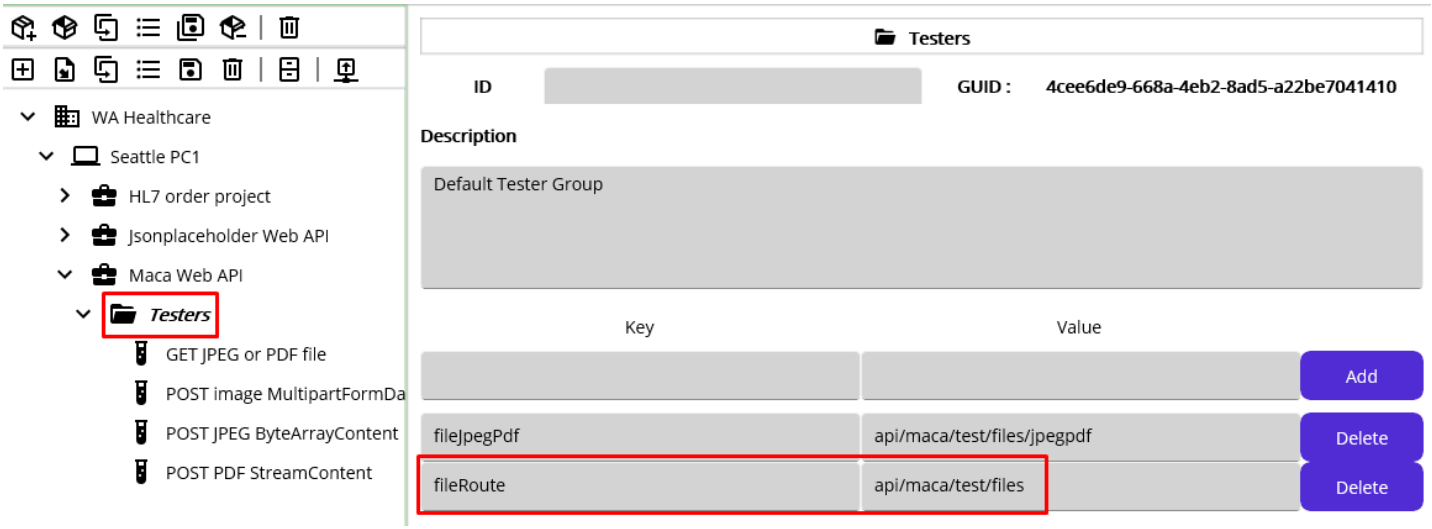


Figure 317.Added fieRoute reference

If you remember our design standard, i.e., 1 group per 1 route, a new ServiceGroup needs to be created. Since we are not implementing real service, and the service is part of testing, we can reuse the Testers group by adding the new route to share that in the containing services. For the test, keep the new route here and save the Testers model.

Back to the service and go to the Execution > Steps view. Then click the Web API ( ) button to open the UI. Please select Testers group from the Reference explorer, and set the Route value with recently added fileRoute reference, as shown in Figure 318.

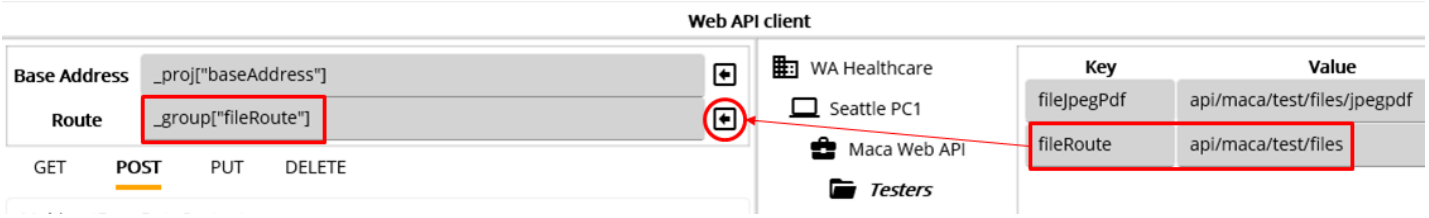


Figure 318.Set the Route with added reference fileRoute

As we upload the image using MultipartFormDataContent, please select that from the HttpContent picker, as shown in Figure 319.

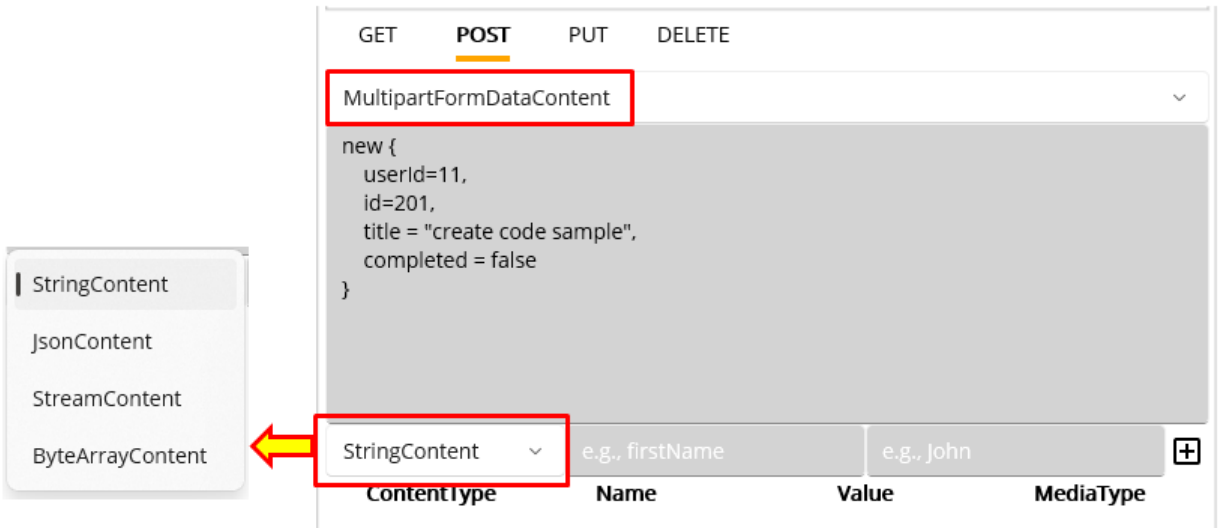


Figure 319.MultipartFormDataContent input view

Once you select the MultipartFormDataContent, you will see the another HttpContent picker right below the anonymous object input pane. As you see, there are 4 HttpContents that you can add to the MultipartFormDataContent. Followed by the route that we

will use, we have to add 4 HttpContents to call the POST, i.e., 3 StringContents and 1 StreamContent(or ByteArrayContent). For now, we are not going to add JsonContent, we will skip that part. So, let's add the content one by one.

Add StringContent to MultipartFormDataContent

Based on the openapi spec in Figure 315, the property locationId needs to be added first.

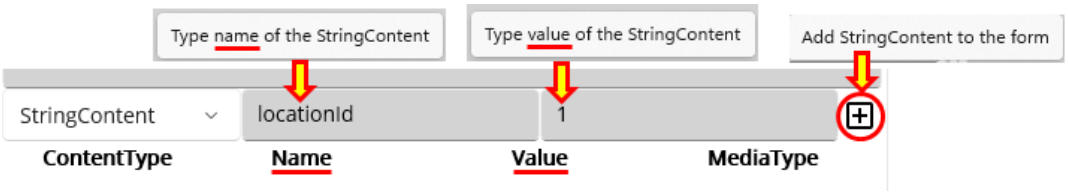


Figure 320.StringContent input area

As you see in Figure 320, the hint messages will popup when you hover your mouse over two input ares. In addtion, the added contents area has associated columns, Name and Value. Compare this to the OpenApi spec, you will notice that the name here should be the property name of the spec. Followed by this rule, you can add other properties in the same way. For now, click the "Add StringContent to the form" button and the added content will look like Figure 312.

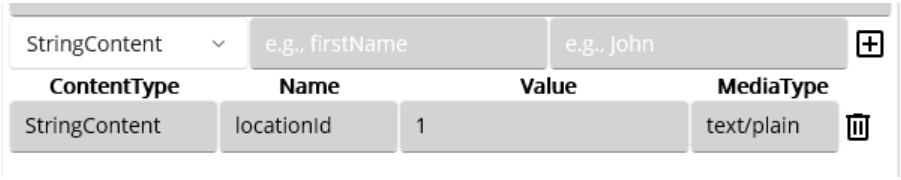


Figure 321.Added StringContent to MultipartFormDataContent

One thing to note here is the MediaType that was auto-populated with "text/plain." As you will see later, the UI will auto-populate for the selected HttpContent. Because the Web API client UI is designed to provide a template-style code insertion and the file types shown here are limited, you need to modify the inserted code in the code editor in order to submit different MediaType based on the requirement. But once you finish this tutorial, we believe you will easily identify where and what needs to be modified.

Let's add two more properties, patientId and dateTaken as shown in Figure 322.

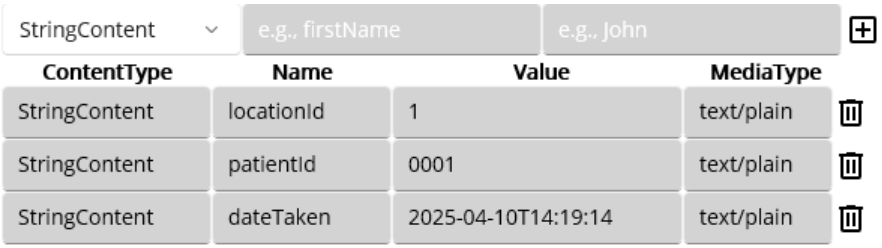


Figure 322. Added 3 StringContents

As you see, all the Name fields are populated with the propertie names in the OpenAPI spec. One minor thing about the value is the patientId's value 0001. Although it looks like a number and needs to be typed as 1, the value there must be 0001. In a certain system or in a file, e.g., DICOM file, the id value sometimes is assigned with prefix 0 for some reason. Likewise, the Maca Web API requires that property as string, the value will be kept in the server as "0001." So, whatever the value the system or file maintains, we have to capture the value as is unless the receiving application requires the numeric value. We will cover this with an example later. Lastly, the dateTaken property's value is set as "2025-04-10T14:19:14," which is ISO 8601. Now, let's add an image data.

Add StreamContent or ByteArrayContent to MultipartFormDataContent

Since the server expected IFormFile for the file, we can submit either StreamContent or ByteArrayContent for the image file. Although the POST is not well described, the method requires only jpeg content type, i.e., "image/jpeg" or "image/jpg." All other files will be returned as 400 Bad Requet. For the first submission, choose the StreamContent, as shown in Figure 323.

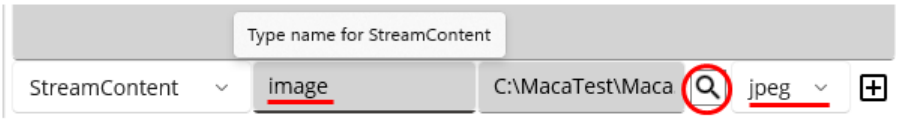


Figure 323.StreamContent input area

As we saw earlier, what we have to do is to locate the file to upload. After selecting “jpeg” from the picker, set the file location using the magnifier button. And the file location informatin will be used to create a StreamContent. Based on the preceding steps, the following Figure 324 show the comparison between the API spec and the implemented Contents in the MultipartFormDataContent.

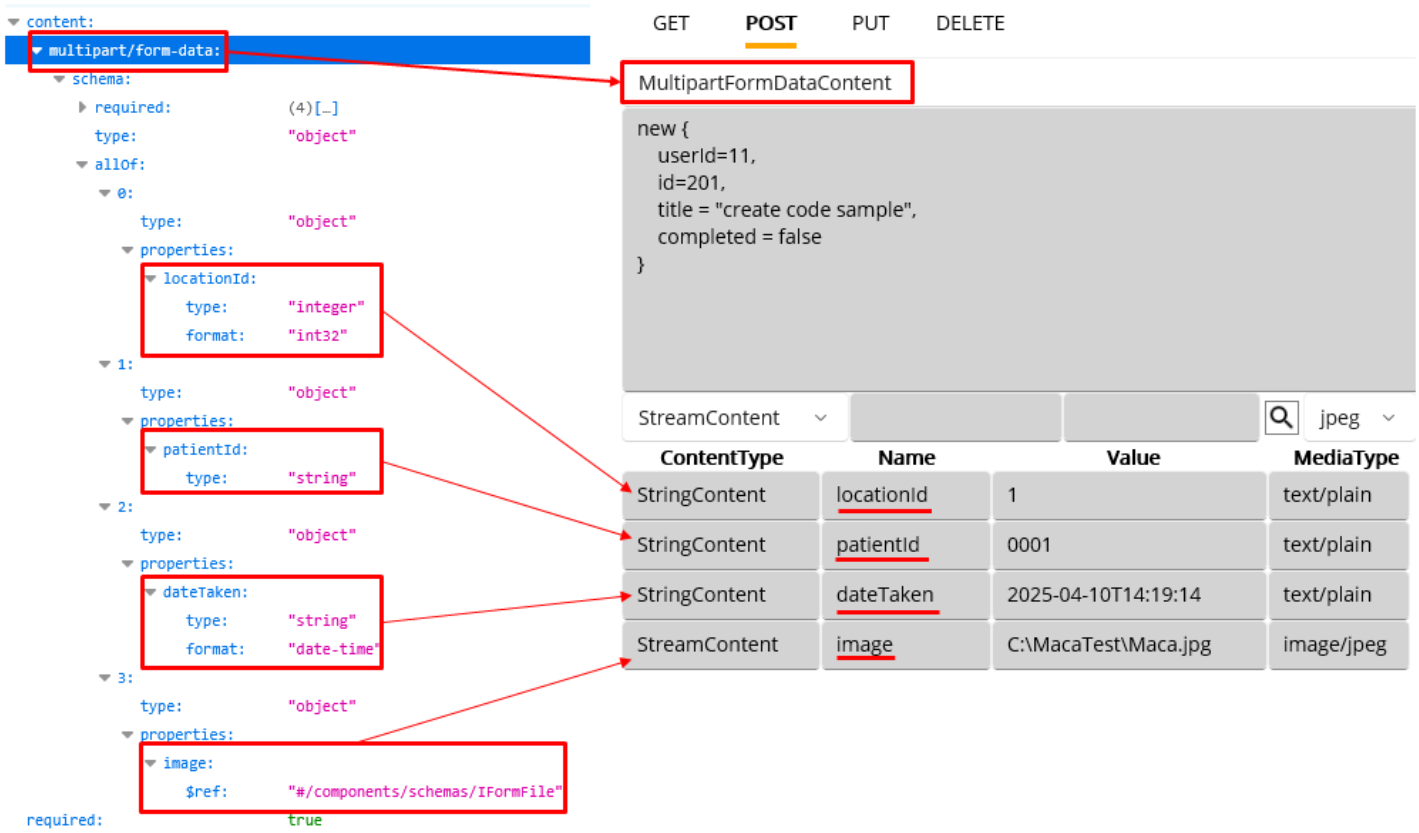


Figure 324.API spec and MultipartFormDataContent

Note. Although the OpenAPI spec assigned the index to each property, we don’t have to add them sequentially. As long as we provide all required properties, the order is not considered.

With this configuration, let’s test the POST. Please click the Execute button.

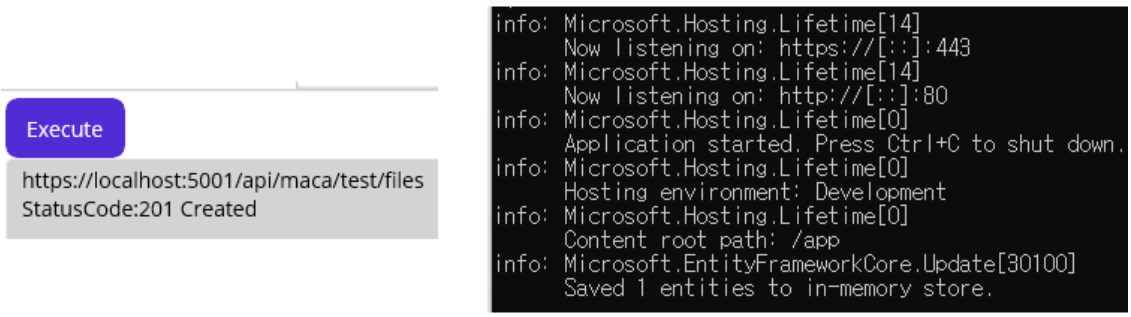


Figure 325.Result of POST with MultipartFormDataContent

As you see, the server returns 201, and the server shows 1 entity is saved(uploaded) to the im-memory database. To test a little more, please click the Execute buttong again, i.e., upload the same file again.

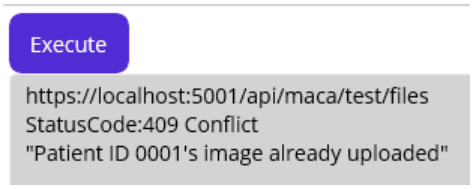


Figure 326.Response for a file already uploaded

At this time, we saw the conflict message that says the image is already uploaded. By design, the POST in the "api/maca/test/files" doesn't allow reupload or process POST request as PUT, i.e., replace or update the existing image. As the API is for simple file uploading test, it doesn't have a DELETE method to delele file. So, please call this not too many or stop the container.

As we saw the uploading the image using MultipartFormDataContent was successful, let's insert the code to the code editor. Click "Insert code and close" button.

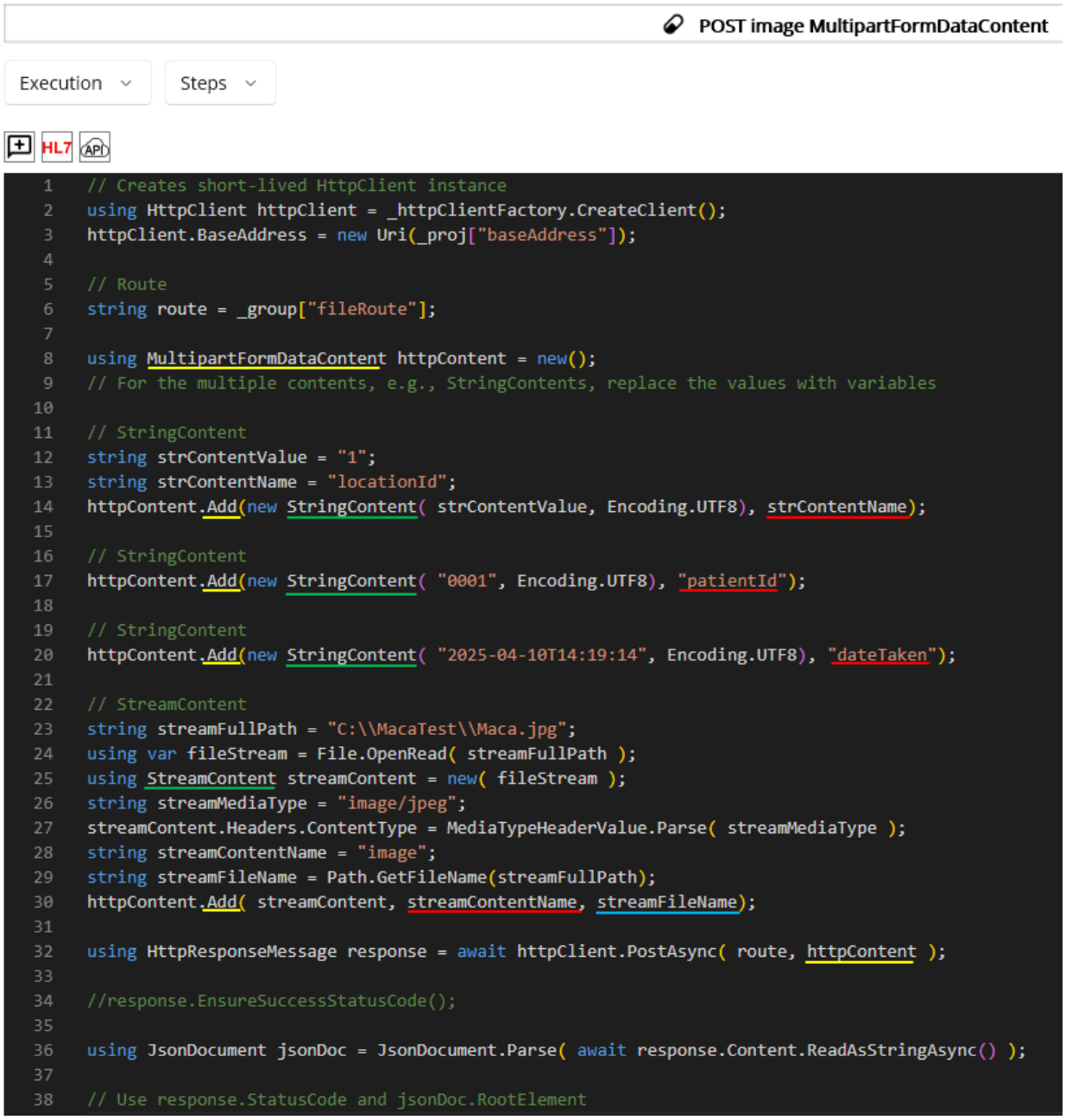


Figure 327.Inserted MultipartFormDataContent code

As you see in Figure 327, there are several parts to check. As we did, in every creation of the HttpContent, e.g., StringContent, we added that to the MultipartFormDataContent using Add method. Then we specified the arguments of the method. For example, the 2nd argument of the Add must be the property name as you see the undersocred value in blue. And the last content, i.e., SreamContent, is what we used before, so the usage of it will be the same. But in the case of StreamContent, the Add method requires the 3rd argument, which is the file name and extension. So we had to get that value and put it in the 3rd location of the Add method. It looks a bit lengthier than the other service's code, but the actual code here would be easy to follow.

Since there is no message handling code, we can add few lines of code to save the returned message, as shown in Figure 328.

```
using HttpResponseMessage response = await httpClient.PostAsync( route, httpContent );

if( (int)response.StatusCode == 201 )
{
 result.ResultMessage = response.StatusCode.ToString();
}
else
{
 result.TransactionResult = TransactionResults.Error;
 result.ResultMessage = await response.Content.ReadAsStringAsync();
}
```

Figure 328.Adding returned message to RonResult

The code simply saves the status code in case of 201. Otherwise, saves the exception message and the result will be set as an Error, i.e., the message will be shown on the Error Message popup.

As a last step, we need to update the patientId to "0002" to make sure the successful upload after we deploy the service. **Update the 2nd StringContent's "0001" value with "0002."** Then click the Verify Service ( ) button to see if there's any issue.

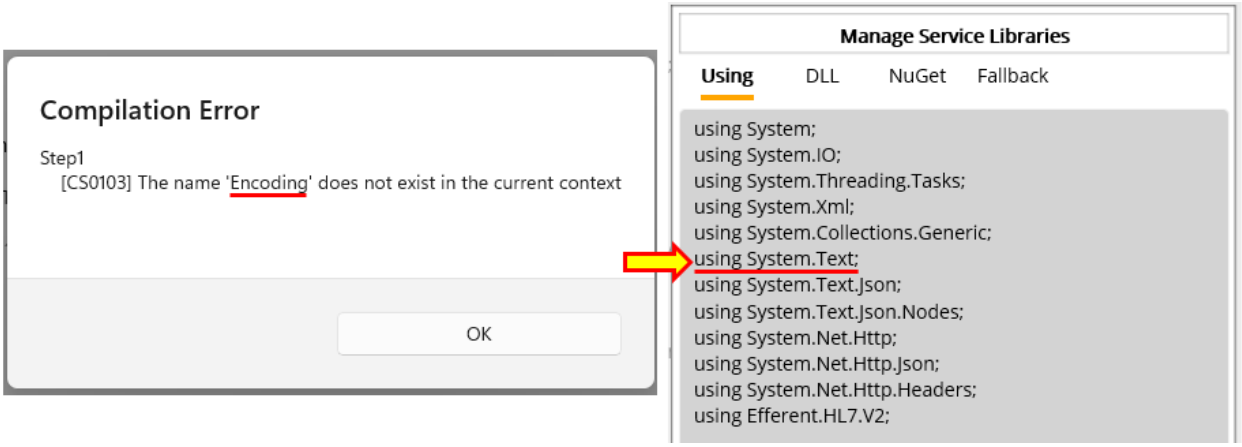


Figure 329.Missing System.Text namespace

Note. By design, Maca doesn't add System.Text namespace in the default Using area to make sure the user practice the UI, especially the Using area. It could be added in the later version, such as Beta, but for the prototype, we will keep omitting that for the instruction purpose. So, you will keep seeing the same error message when you use the objects in the System.Text namespace, such as StringBuilder, in the Code editor.

Once you update the value and the verification finishes without an error, load the service. As we created the service to run every minute, when we deploy the service, the first round of the execution will be 201. So, let's load and then deploy the service. When the deploy was successful, siwtch to the Management workstation and go to Observer > Transaction on the Instance tabe. Select Today for the Requested option. Then click the Retrieve button to see the result of the execution.

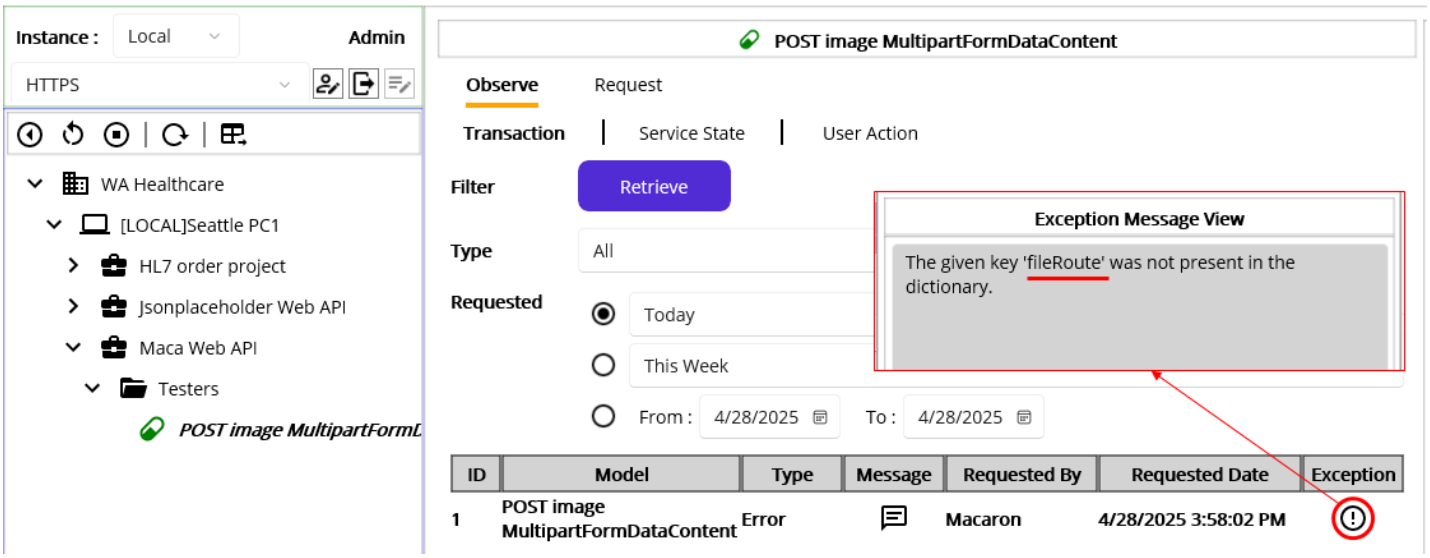


Figure 330.Missing fileRoute reference in Testers group

Unlike our expectation, the service was marked as Error and the Exception message shows that "fileRouter" is not found. Yes, we updated the References by adding "fileRoute" value in the Testers but never deployed it. So, the deloyed service pointed at the missing Reference, and the transaction was marked as Error, as shown in Figure 330.

To resolve this, we can deploy the updated Testers model. But at this time, we will not stop the service, i.e., keep the service running. Then we will keep tracking the transactions to see what will happen after the Testers deployment. **Please load the uploaded Testers only.** When the poup shows up, **click the No button** to load the Testers group only to avoid unexpected service update on the server.

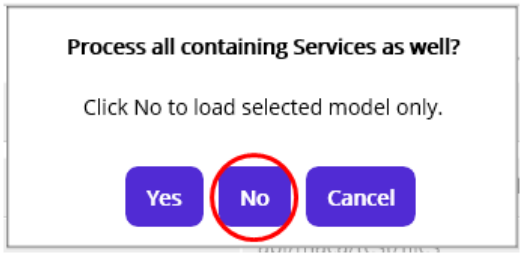


Figure 331.Load the Testers group only

Switch to the Management workstation. Click Source tab and refresh the models to see the updated Testers group. Then deploy it. As we did on the Development workstation, **please click No to deploy the Testers group only,** as shown in Figure 332.

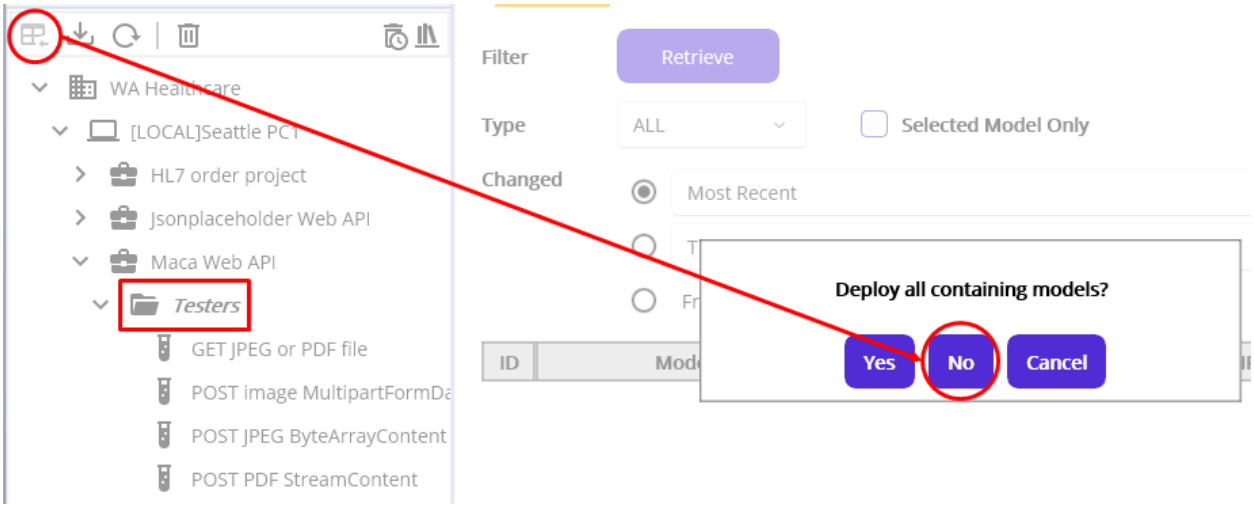


Figure 332.Deploy the Testers group only

Once the deployment is done, click the Instance tab. Then click the Testers group to see the References are updated, as shown in figure 333.

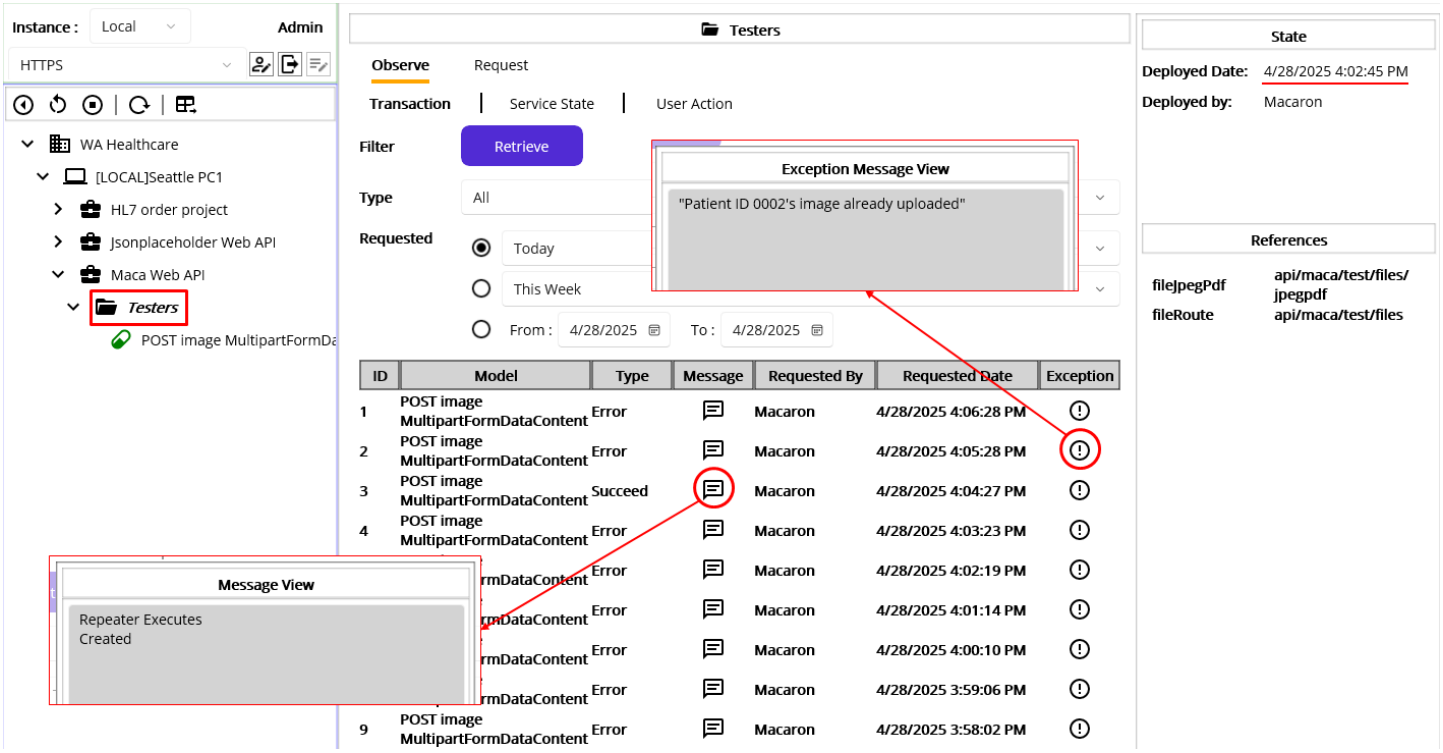


Figure 333.Updated Testers group and the transactions of the containing service

As you see, once the Testers group deployed, the next execution of the service was finished with Succeed, and the Message shows "Created" as a result. And as we expected, the consecutive execution resulted in Error with "Patient ID 0002'S image already uploaded" message.

Although we just implement the code with the happy path, you can test the service by modifing inserted code like submit the requeist without file or replacing the property name with other values. As we are not going to test the service thoroughly, we recommend you to keep playing with the service to see what happens.

Noteworthy points in the preceding test

We made a change in the Group model, especially the References, to support multiple routes with one group. We could successfully test on the Web API client UI with the modifined Group but forgot to upload the group model. It could be a common situation while we developing services along with the group modles. Once you touch the group models, such as Domain or project, please make sure the model is redeployed as well. But the important point is **to load and deploy the modified group model only** to avoid any unexpected update on the containing services. Since Maca is a prototype version, the internal functionality is not fully implemented to cover all scenarios. So, it is recommended to follow the tutorial to get the expected results.

Implementing patient image upload

While we practicing the file uploading, WA Healthcare IT team finally shared the spec of the services. They decided to accept an image file along with JsonContent that contains the patient information. What they provide is the same as the one of the Maca Web API with a route "**api/maca/files,**" as shown in Figure 334. So, you can use its POST method to implement the requirement.

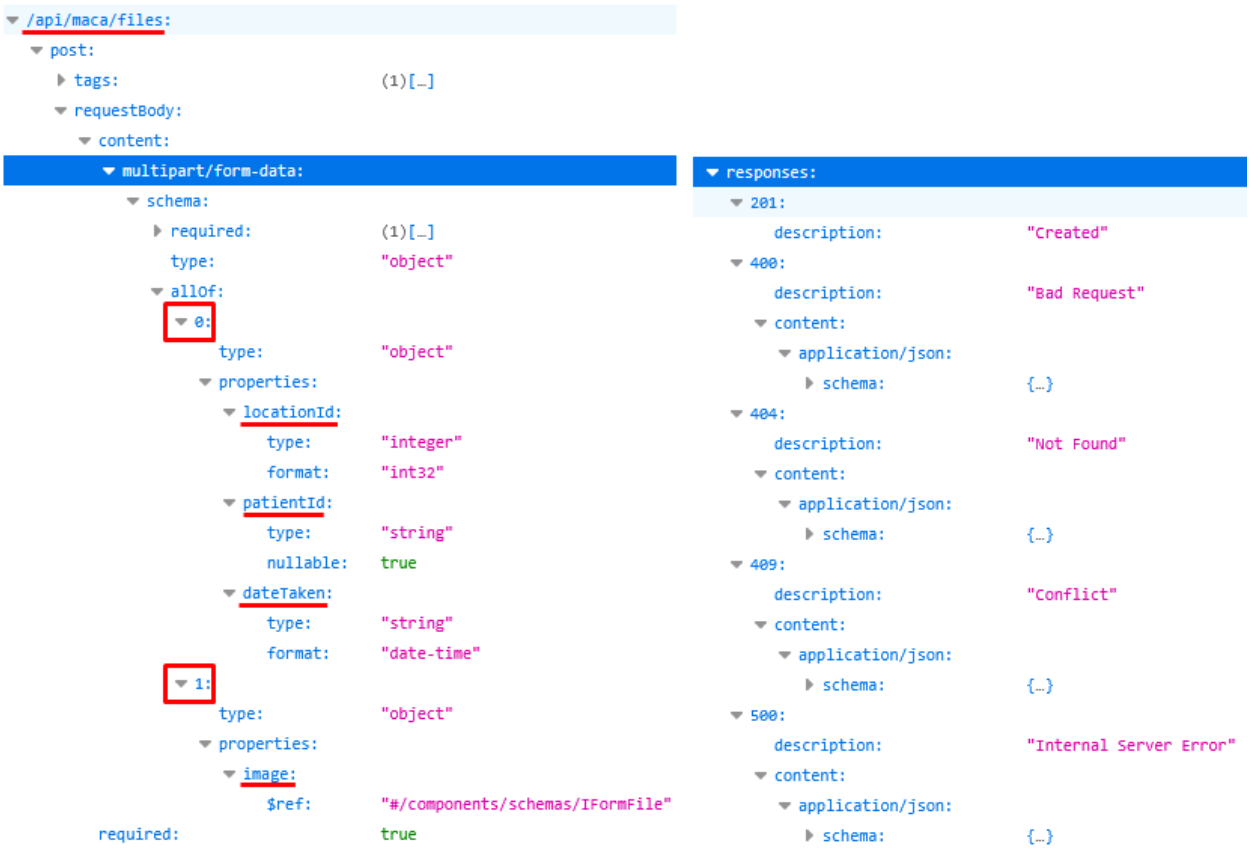


Figure 334.MultipartFormDatacontent with JsonContent and StreamContent(ByteArrayContent)

One thing that is not clear in the spec is the 1st property with index 0, which is an object type. Based on the code, it is set as a class, usually called DTO(Data Transafer Object), and defined as "patient." Typically, the DTO is auto-mapped with a JSON, we can consider the object a JsonContent. For your reference, the DTO could look like following Figure 335.

```
public class PatientDto
{
   public int LocationId { get; set; }
   public string PatientId { get; set; }
   public DateTime DateTaken { get; set; }
}
```

Figure 335.PatientDTO for the JsonContent

Based on that, we can set the first property as JsonConten with a name "patient." Then the 2nd property is a file that we can submit as StreamContent or ByteArrayContent. Now, we know what the server requires, the next step is extracting that information and data from the DICOM file that we handled in the DICOM Reader service. So, we can modify the DICOM Reader service to implement that requirement. But, **instead of the direct modification, we can clone the service to the Testers group and modify that service**. By doing so, we can freely test the service without affecting production level service. Please clone the service, as shown in Figure 336.

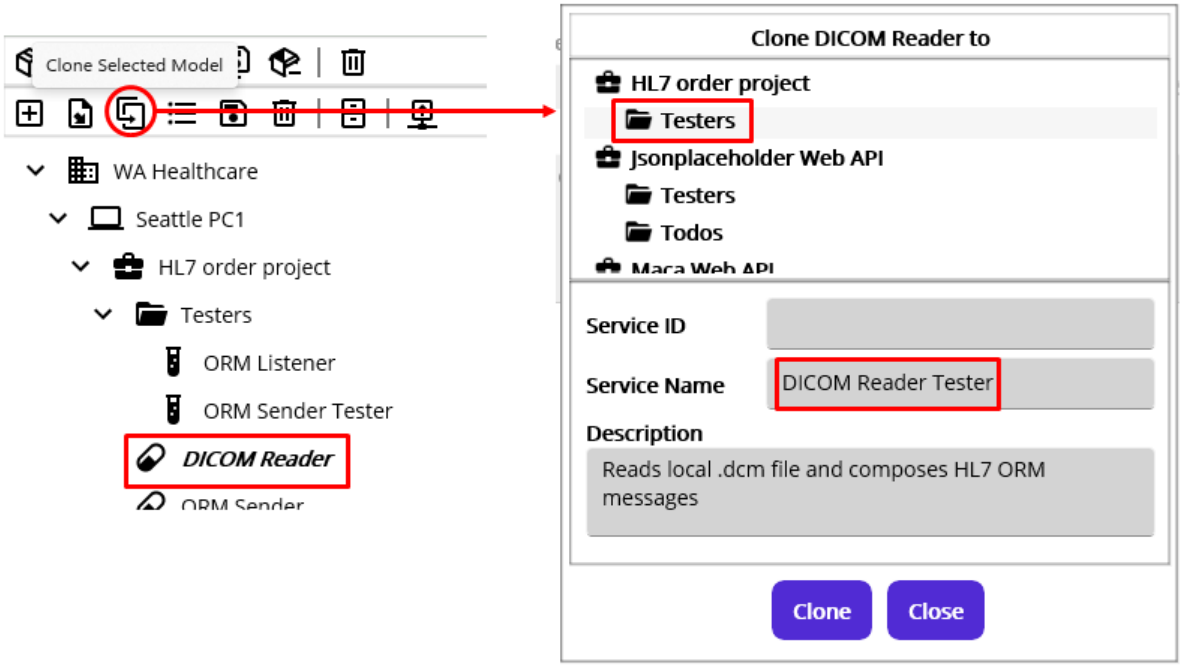


Figure 336.Clone DICOM Reader to Testers group

Once you clone the service, go to Execution > Steps.

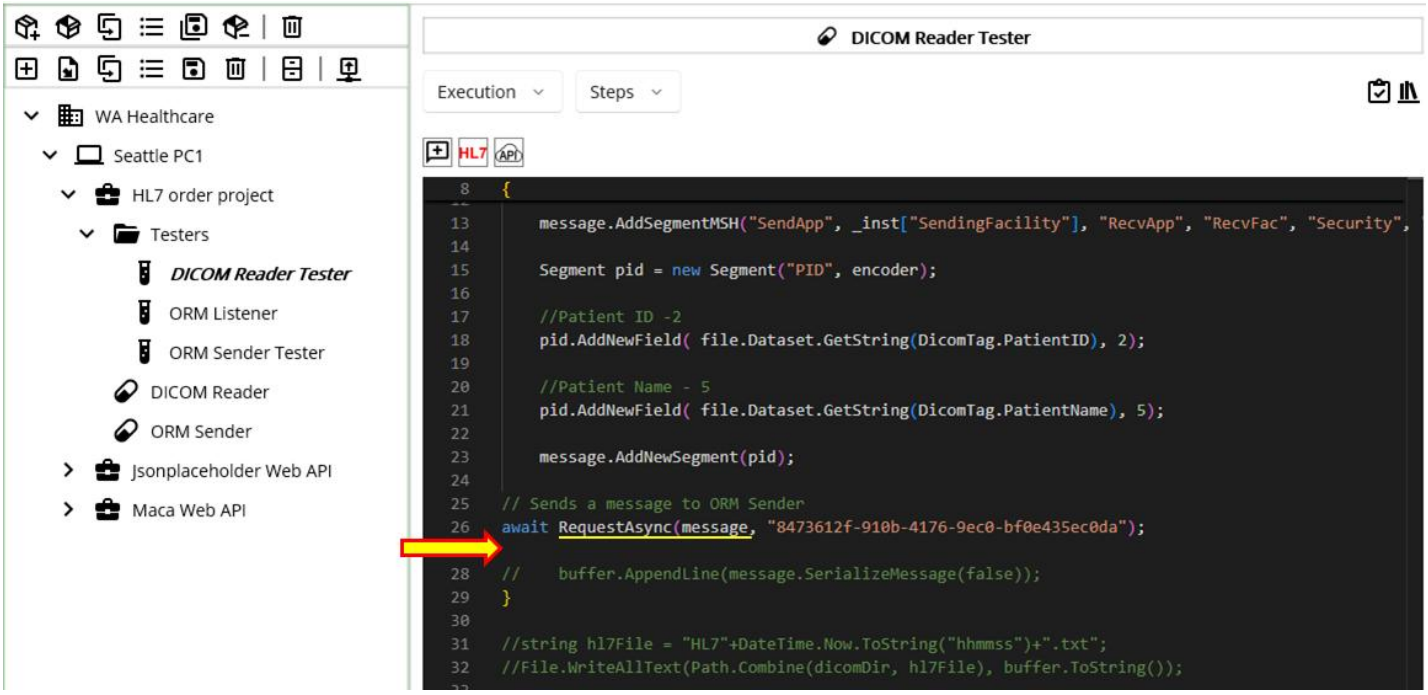


Figure 337.Where to insert the Web API call code

In the existing code, we extracted some values from the DICOM and called the ORM Sender service. Based on that process, we can insert Web API calling code right below the RequestAsync method, as shown in Figure 337. Since we will insert the code that might be more than 20 lines, we may add extra comment lines to make the inserted code identifiable like below Figure 338.

```
// Sends a message to ORM Sender
await RequestAsync(message, "8473612f-910b-4176-9ec0-bf0e435ec0da");

#region "Web API Call Code"
// place the cursor in the below line to insert the code

#endregion

//    buffer.AppendLine(message.SerializeMessage(false));
}
```

Figure 338.Placeholder lines to insert the code

As the commented line says, place the cursor inside the region to prevent the code from being inserted in the middle of the unwanted line.

Test file uploading

Previously, we located the line to insert Web API calling code. Now, let's open Web API client UI and test the given service to upload sample image to the server. Click the API button to open the UI.

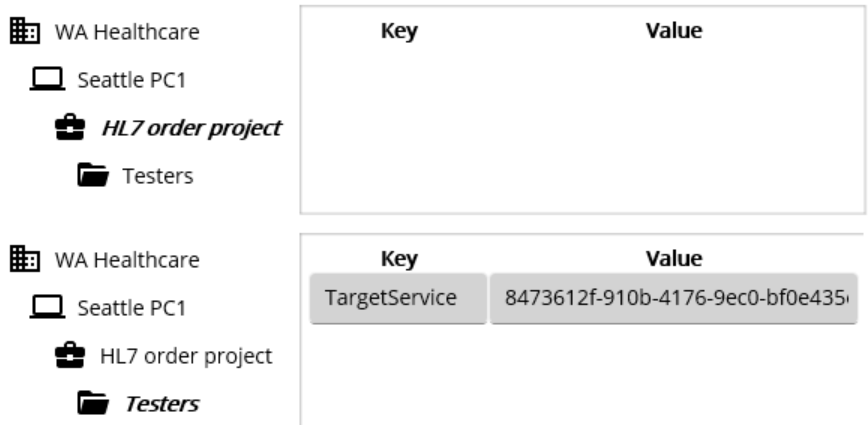


Figure 339.No references related to the Maca Web API

When you open the UI, you will face a cumbersome situation that there is no Maca Web API related references that you can add to the target destination. So, whenever you open this UI, you have to type base address and route manually unless you add them in the project or group. **For this test, we will keep typing the base address and route manually every time.** We will explain later why we take this approach with examples. But for now, please write down them in the notepad or similar text editor and paste them here.

Once you type the base address and route, select MultipartFormDataContent. Then select the JsonContent from the subsequent HttpContent picker, as shown in Figure 340..

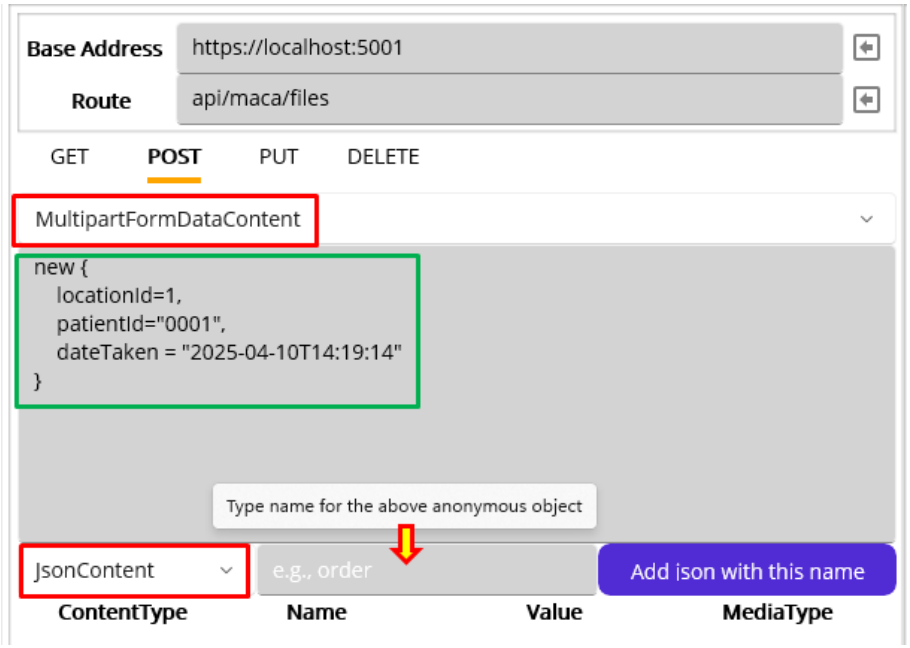


Figure 340.JsonContent input area in the MultipartFormDataContent

Among other things, the anonymous object type panel is the key part of the JsonContent input area. In the earlier examples with the Jsonplaceholder Web API, we used the anonymous object to map values to Todo class. If you skipped that part or need a brush-up, please check Figure 253. In the same vein, the anonymous object can be mapped with the DTO that the server requires, as shown in Figure 341.

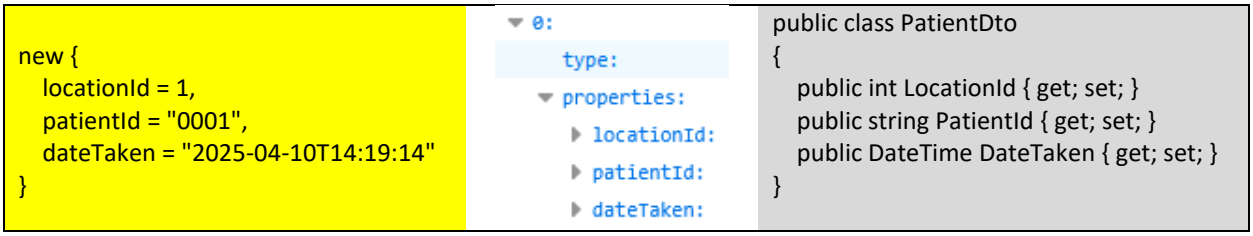


Figure 341.Anonymous object, api spec, and PatientDto class

As you see, it is quite important to know what the server requires to implement the requirement. So, please keep checking your client's specificiation to see if there is any unclear or missing information.

Previously, we clarified that the JsonContent is named with "patient," so, type "patient" in the name area and click the "Add json with this name" button to add. Then add jpeg image as StreamCoontent, as shown in Figure 342.

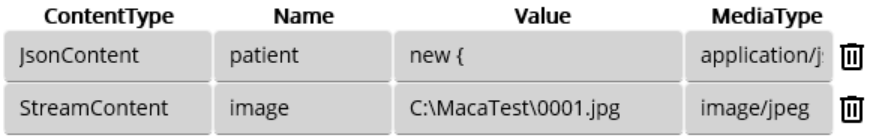


Figure 342.Added JsonContent and Streamcontent

So far, we added all required contents to the MultipartFormDatacontent. Now, click the Execute button to execute POST method.

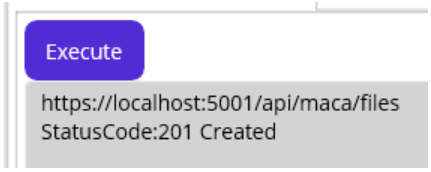


Figure 343.Success respose from the server

As you see in Figure 343, the POST request was processed successfully in the server. By design, the POST method doesn't allow image update. If you click the Execute button again, you will see the following error message, as shown in Figure 409.

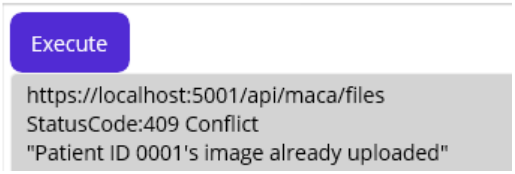


Figure 344.409 message if the same patient image is submitted

If you keep running the service with the same values, i.e., in the JsonContent, the server will keep returning 409 message.

Note. The API doesn't have PUT or DELETE in the "api/maca/files" route to make the API simple. To upload more images, please modify JsonContent or restart the container.

In the preceding test, we saw the image was successfully uploaded on the server. Now, let's insert the code. Please click "Insert code and close" button. Then you will see the auto-generated code was inserted, as shown in Figure 345.

```
#region "Web API Call Code"
// Place the cursor in the below line to insert the code
// Creates short-lived HttpClient instance
using HttpClient httpClient = _httpClientFactory.CreateClient();
httpClient.BaseAddress = new Uri("https://localhost:5001");

// Route
string route = "api/maca/files";

using MultipartFormDataContent httpContent = new();
// For the multiple contents, e.g., StringContents, replace the values with variables

// JsonContent
var jsonPayload = new {
    locationId = 1,
    patientId = "0001",
    dateTaken = "2025-04-10T14:19:14"
};
string jsonContentName = "patient";
httpContent.Add(JsonContent.Create( jsonPayload ), jsonContentName);

// StreamContent
string streamFullPath = "C:\\MacaTest\\0001.jpg";
using var fileStream = File.OpenRead( streamFullPath );
using StreamContent streamContent = new( fileStream );
string streamMediaType = "image/jpeg";
streamContent.Headers.ContentType = MediaTypeHeaderValue.Parse( streamMediaType );
string streamContentName = "image";
string streamFileName = Path.GetFileName(streamFullPath);
httpContent.Add( streamContent, streamContentName, streamFileName);

using HttpResponseMessage response = await httpClient.PostAsync( route, httpContent );

//response.EnsureSuccessStatusCode();

using JsonDocument jsonDoc = JsonDocument.Parse( await response.Content.ReadAsStringAsync() );

// Use response.StatusCode and jsonDoc.RootElement

#endregion
}
```

Figure 345.Inserted auto-generated MultipartFormDataContent code

Based on the HttpContents that we inserted, what we need to configure can be defined as follow:

1. Set JsonCotent's properties (**red box**) with DICOM tag values.
2. Set ByteArrayContent's data with DICOM's internal image data.

Map DICOM data to anonymous object

As we defined what needs to be read, let's check the property one by one.

Read locationId

If you still rember, we set the SendingFacility in the Instance References. So, we can use that reference to set locationId, as shown in Figure 346.

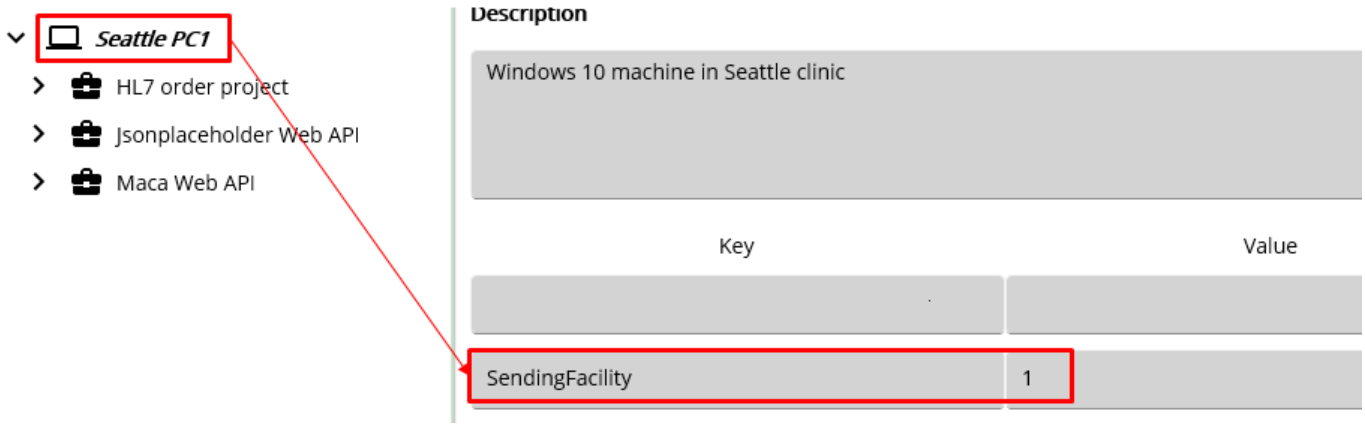


Figure 346.SendingFacility reference for locationId

Read patientId

For patient id, original code already had that value like below, so we can reuse that part.

```
//Patient ID -2
pid.AddNewField( file.Dataset.GetString(DicomTag.PatientID), 2);
```

Read dateTaken

Similar to the patientId, we can use the kewords in the DICOM dictionary (https://github.com/fo-dicom/fo-dicom/blob/development/FO-DICOM.Core/Dictionaries/DICOM%20Dictionary.xml) to get dateTaken related data. In our case, we need study date and study time, as shown in Figure 347.

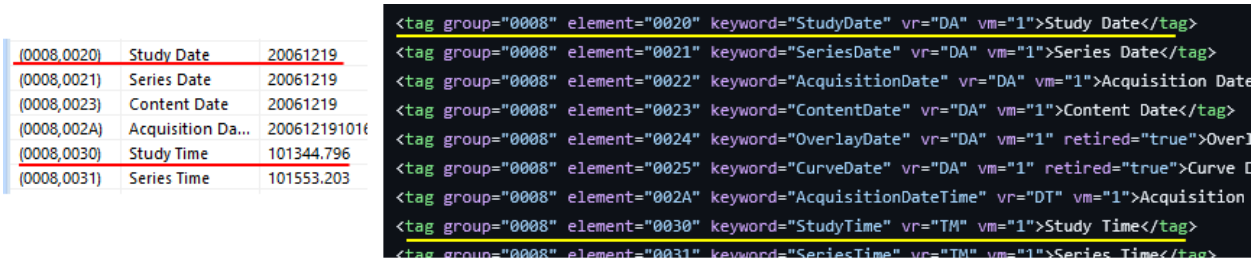


Figure 347. Study date and time in the DICOM viewer(Left) and the DICOM dictionary(Right)

Map all values to the properties

Using above 3 values, we can set the jsonPayload's properties, as shown in Figure 348.

```
// JsonContent
var jsonPayload = new {
   locationId = int.Parse(_inst["SendingFacility"]),
   patientId = file.Dataset.GetString(DicomTag.PatientID),
```

```
    dateTaken = file.Dataset.GetDateTime(DicomTag.StudyDate, DicomTag.StudyTime)
};
```

Figure 348.Map 3 properties with DICOM tag values

All codes are straightforward to follow. Thankfully, the fo-dicom converts two tags to the ISO 8601 format, e.g., "2025-04-10T14:19:14," as we mentioned earlier. If your client's application requires other format, please set that value accordingly.

Read JPEG image

Unfortunately, we can't read the image directly from the Tag using existing fo-dicom library. In addition, the POST method requires JPEG media type. We need extra steps to extract the iamge from the DICOM file. The first step is to install image handling extension from NuGet. So, click NuGet tab and search and download **fo-dicom.Imaging.ImageSharp** and **SixLabors.ImageSharp**, as shown in Figure 349.

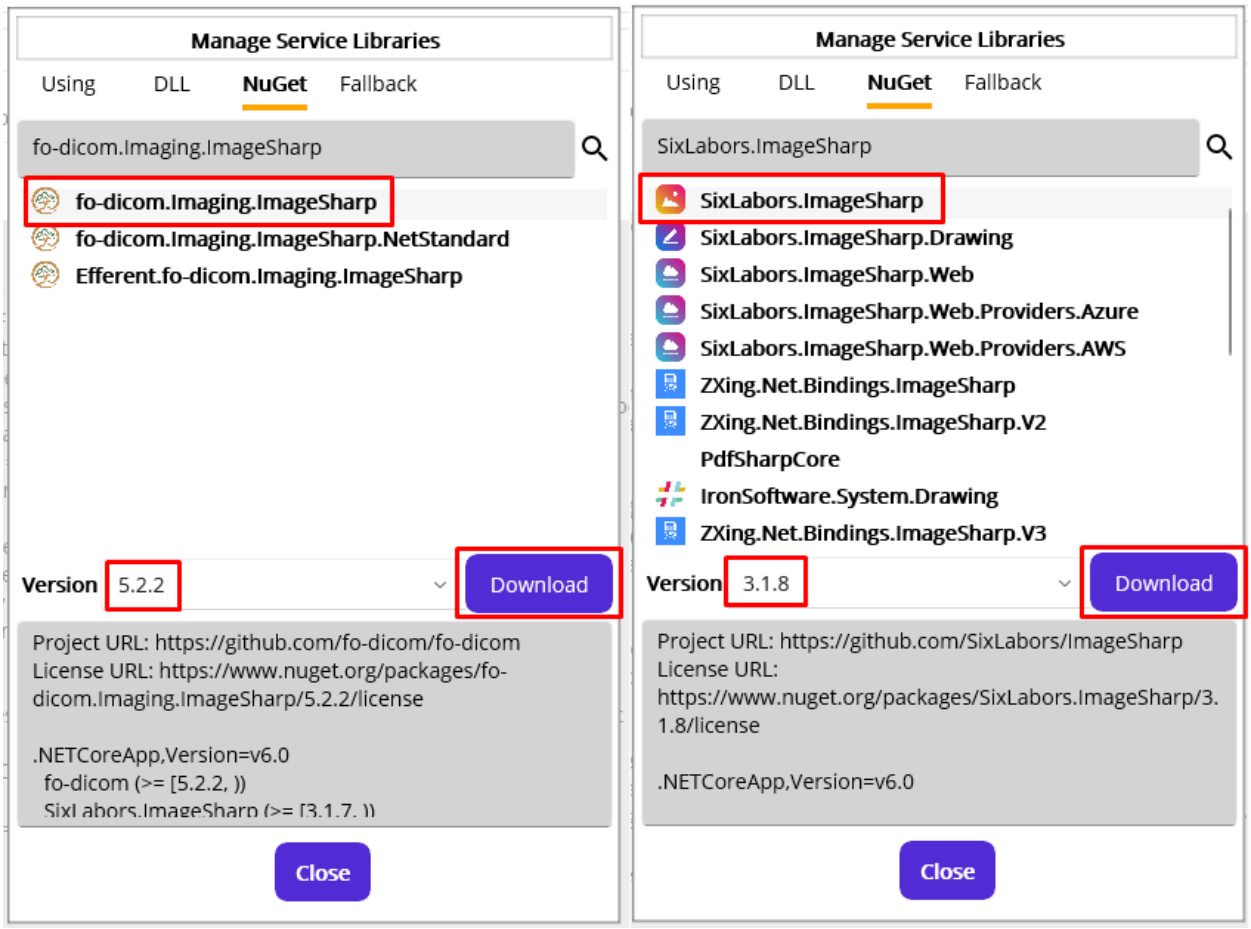


Figure 349.Download fo-dicom ImageSharp extension

Then, **add the downloaded DLLs in the DLL tab**, as shown in Figure 350.

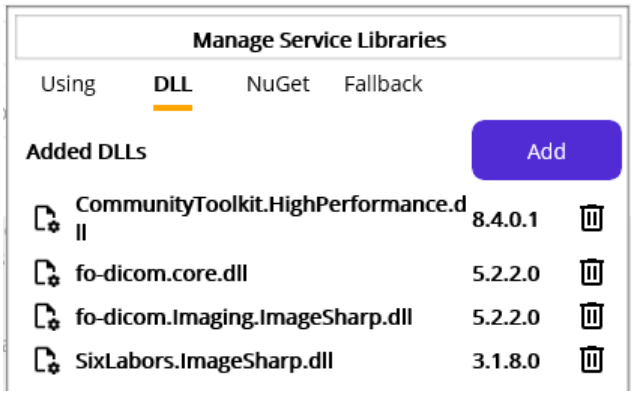


Figure 350.Added DICOM image related DLLs

Next, add the namespaces "**using FellowOakDicom.Imaging;**" "**using SixLabors.ImageSharp;**" "**using SixLabors.ImageSharp.Formats.Jpeg;**" "**using SixLabors.ImageSharp.PixelFormats;**" to be referenced in the Code editor, as shown in Figure 351.

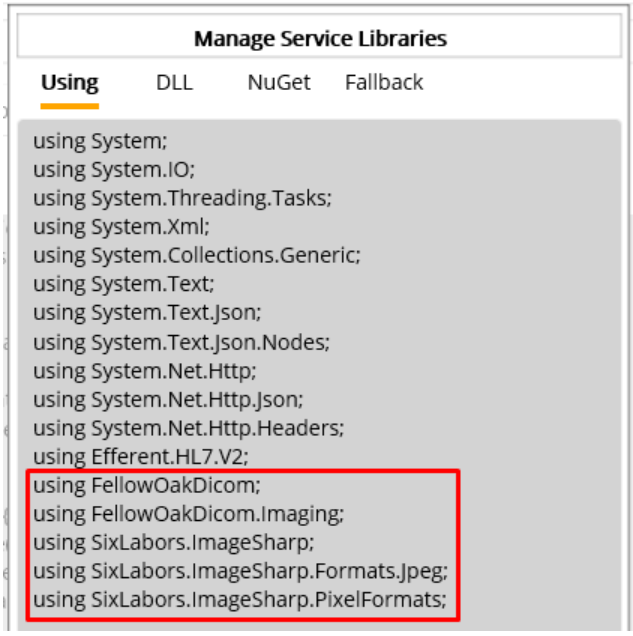


Figure 351.All DICOM data handling namespaces

Once you add the namespace, we need to modify inserted StreamContent code, which has hard-coded values. Replace that code with the JPEG reading code, as shown in Figure 352.

```
// StreamContent
DicomImage dcmImage = new ( dcm );
IImage image = dcmImage.RenderImage();
using var rgba32 = image.LoadPixelData<Rgba32>( image.AsBytes(), image.Width, image.Height );
using var stream = new MemoryStream();
rgba32.Save(stream, new JpegEncoder { Quality = 100 });
stream.Position = 0;

using StreamContent streamContent = new( stream );
string streamMediaType = "image/jpeg";
streamContent.Headers.ContentType = MediaTypeHeaderValue.Parse( streamMediaType );
string streamContentName = "image";
string streamFileName = file.Dataset.GetString(DicomTag.PatientID) + ".jpeg";
httpContent.Add( streamContent, streamContentName, streamFileName);
```

Figure 352.ByteArrayContent with JPEG image data

Extracting image is little bit tricky here but we are not going to go over line by line at this time. So, please check the SixLabors website(https://docs.sixlabors.com/articles/imagesharp/pixelbuffers.html) and https://fo-dicom.github.io/dev/v5/api/FellowOakDicom.Imaging.DicomImage.html for the detail.

Save the result of the response

As we did in the other services, we can save the result of the response message to the RonResult. So, we can resue the code in the other POST service, as shown in Figure 353.

```
if( (int)response.StatusCode == 201 )
{
  result.ResultMessage = response.StatusCode.ToString();
}
else
{
  result.TransactionResult = TransactionResults.Error;
  result.ResultMessage = await response.Content.ReadAsStringAsync();
}
```

Figure 353.Save response message th the RonResult

In our case, if you let the service keep running, the first execution will upload the image. And the subsequent calls will be 409 conflict.

Handle the DICOM after uploading

We do not remove or archive the DICOM after procssing whether the result of POST is success or fail. You can add post-processing code accordingly.

Deploy implemented test service

The following Figure 354 shows the implemented code in the region “Web API Call code.” But please note that calling ORM Sender is commented. Since we didn’t add any error handling code when calling the ORM Sender, the sebsequent code won’t be executed if the ORM Sender is not running. So, we comment the line and let the rest of the code execute.

```
var dicomDir = @"C:\Maca\Dicoms";

StringBuilder buffer = new ();

HL7Encoding encoder = new HL7Encoding();

foreach (var dcm in Directory.GetFiles(dicomDir, "*.dcm"))
{
    var file = await DicomFile.OpenAsync(dcm);

    Message message = new Message();

    message.AddSegmentMSH("SendApp", _inst["SendingFacility"], "RecvApp", "RecvFac",
     "Security", "ORM^O01", "20240512", "T", "2.3.1");

    Segment pid = new Segment("PID", encoder);

    //Patient ID -2
    pid.AddNewField( file.Dataset.GetString(DicomTag.PatientID), 2);

    //Patient Name - 5
    pid.AddNewField( file.Dataset.GetString(DicomTag.PatientName), 5);

    message.AddNewSegment(pid);

// Sends a message to ORM Sender
// await RequestAsync(message, "8473612f-910b-4176-9ec0-bf0e435ec0da");

#region "Web API Call Code"
// Place the cursor in the below line to insert the code
// Creates short-lived HttpClient instance
using HttpClient httpClient = _httpClientFactory.CreateClient();
httpClient.BaseAddress = new Uri("https://localhost:5001");

// Route
string route = "api/maca/files";

using MultipartFormDataContent httpContent = new();
// For the multiple contents, e.g., StringContents, replace the values with variables
```

```
// JsonContent
var jsonPayload = new {
    locationId = int.Parse(_inst["SendingFacility"]),
    patientId = file.Dataset.GetString(DicomTag.PatientID),
    dateTaken = file.Dataset.GetDateTime(DicomTag.StudyDate, DicomTag.StudyTime)
};
string jsonContentName = "patient";
httpContent.Add(JsonContent.Create( jsonPayload ), jsonContentName);

// StreamContent
DicomImage dcmImage = new ( _dicomLocation );
IImage image = dcmImage.RenderImage();
using var rgba32 = Image.LoadPixelData<Rgba32>( image.AsBytes(), image.Width, image.Height );
using var stream = new MemoryStream();
rgba32.Save(stream, new JpegEncoder { Quality = 100 });
stream.Position = 0;

using StreamContent streamContent = new( stream );
string streamMediaType = "image/jpeg";
streamContent.Headers.ContentType = MediaTypeHeaderValue.Parse( streamMediaType );
string streamContentName = "image";
string streamFileName = file.Dataset.GetString(DicomTag.PatientID) + ".jpeg";
httpContent.Add( streamContent, streamContentName, streamFileName);

using HttpResponseMessage response = await httpClient.PostAsync( route, httpContent );

if( (int)response.StatusCode == 201 )
{
  result.ResultMessage = response.StatusCode.ToString();
}
else
{
  result.TransactionResult = TransactionResults.Error;
  result.ResultMessage = await response.Content.ReadAsStringAsync();
}

#endregion
}
```

Figure 354.Implemented DICOM Reader Tester service

Click the Verify Service ( ) button to see if there is any issue with the code. If all looks good, let's deploy the service to see what will be the result look like. But before that, **please stop and then start the Maca Web API container** to make sure we have no image in the server. You can "Ctrl+C" to stop the container on the command prompt. If the Docker Desktop is not running, please start that first.

Once you load and deploy the service, you will se the succeed transaction, as shown in Figure 355.

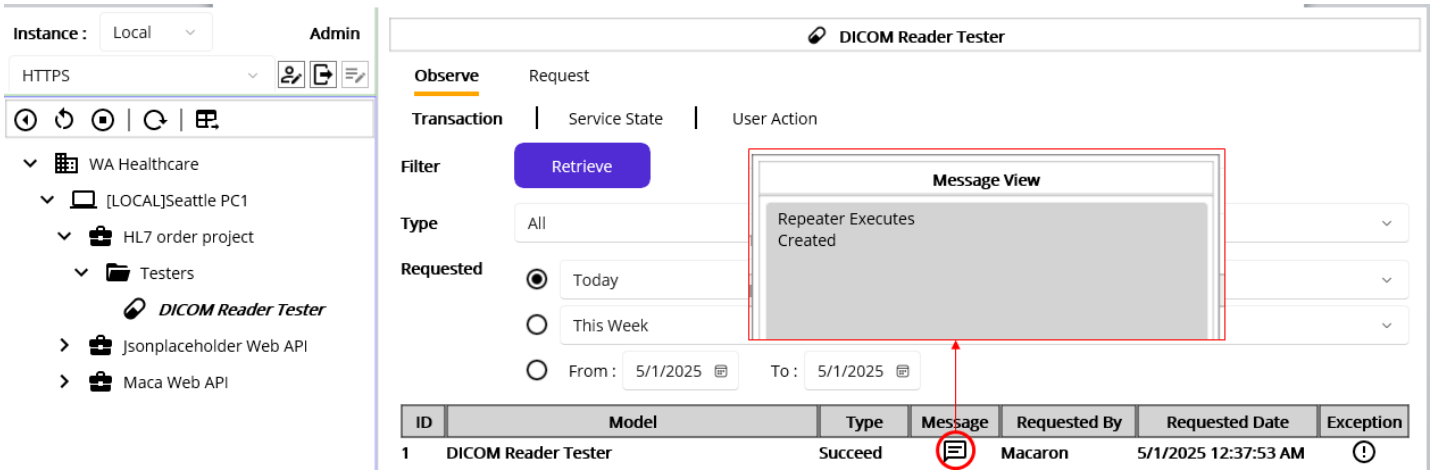


Figure 355.Image upload succeed transaction

If you let the service keep running, i.e., for more than 5 minutes, it will be marked as error, as shown in Figure 356.

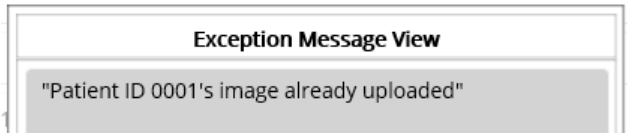


Figure 356.Reupload marked as Error

Finally, we finished the first objective that uploads the patient's image to the server. Although we didn't test the service thoroughly with test cases, we can move on to the next objective because we focus on the core functionality. So, we recommend you to add more testing code with test scenarios, if you have extra time. But before we move on, let's have a short review about the DICOM Reader Tester service.

A service with a dependency

In the preceding section, we added Web API call code inside the DICOM Reader. If you remember what we did to send an ORM message, you will notice that the Web API call code can reside outside as a service as well. Then we can call that service as we called the ORM Sender service. The following Figure 357 shows what can be improved.

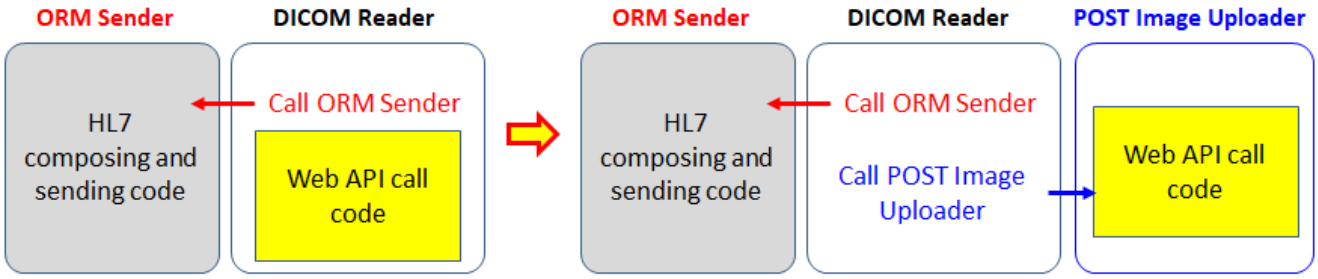


Figure 357.Separated Web API call code

If you look at the code in Figure 354, the code is not only lengthy but also requires extra error handling. In order to test the Web API code or modify it when the API spec is updated, we have to pay extra attention to the other working code that's not part of the Web API call. If we make a separate service that does the Web API call job, it resolves lots of issues.

Benefits of code separation as a servcie

The main benefit of the separation would be that it would be easy to maintain. Since we made the ORM Sender to handle the HL7 exchange job, the DICOM Reader could be kept as-is and be affected none or minimally in case of the ORM Sender update. In addition, we could test ORM Sender in the Management workstation by sending custom HL7 messages. If the TCP/IP handling code is embedded in the DICOM Reader, we had to prepare extra DICOM files to cover lots of test cases. By separating the code, we don't have to worry about the DICOM Reader and can focus on the ORM Sender only.

From the regression perspective, the separation would be a good approach. If the code is embedded in each service, all related services need to be modified and tested one by one in case of logic change or bugs.

Lastly, the service can process multiple requests from the other services. But what we used so far was CodeRepeater that doesn't interact with other services, i.e., can't be callable (requestable). And callable HL7Client service is not suitable to support this requirement, i.e., dedicated to process HL7 message only. In this case, we can use the RequestListener service that listens other services' requests.

Create RequestListener to upload patient images

As we decided to separate the Web API call code, especially the POST method, we need to follow the pattern we applied to the Jsonplaceholder Web API. In our case, the route that we used was "api/maca/files." So, let's create a group named "Files" under the Maca Web API project and add route reference, as shown in Figure

Figure 358.Files route of Maca Web API

Then create a **RequestListener** service named "POST Patient Image Uploader" under the Files group, as shown in Figure 359.

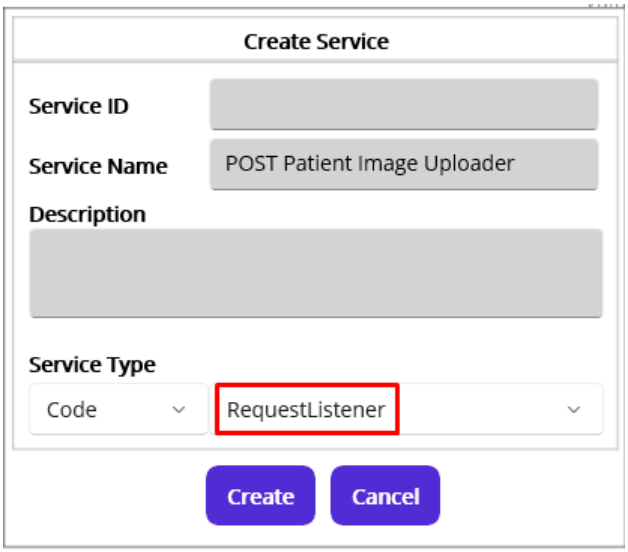


Figure 359.Create a RequestListener service

Once you create the service, go to Settings > Config view, as shown in Figure 360.

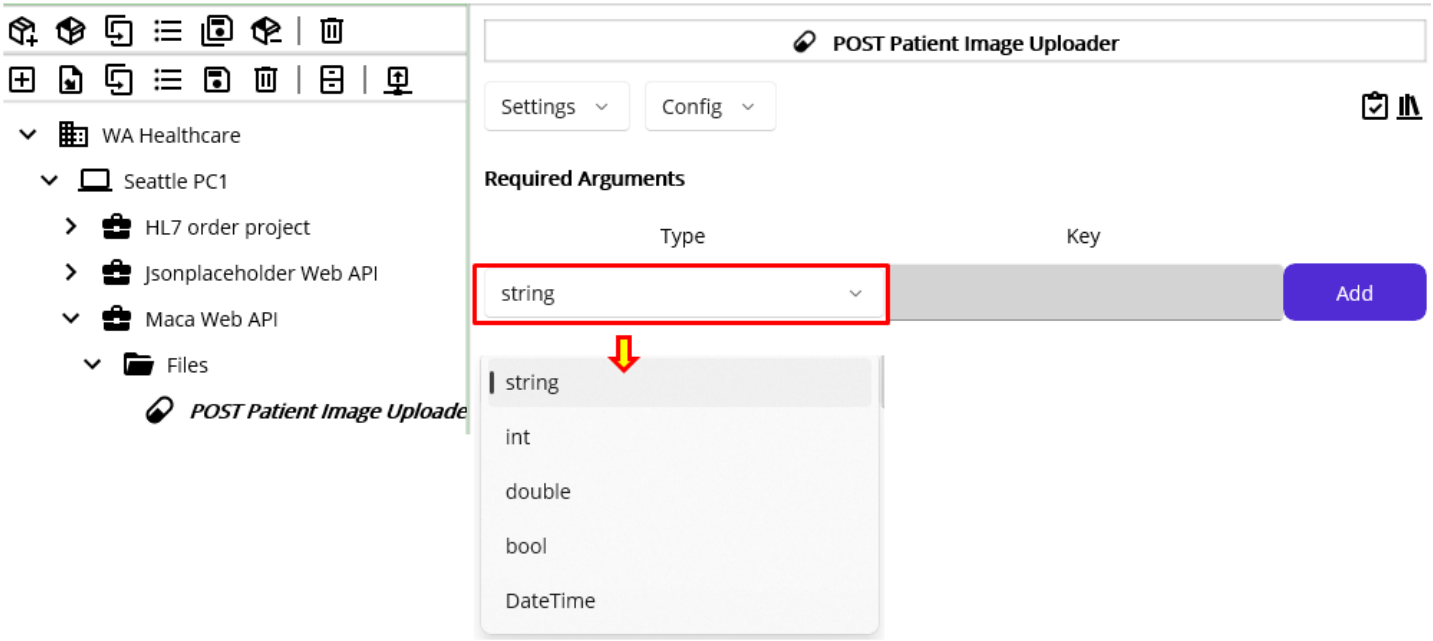


Figure 360.Required Argument input area

As you see, you can set what POST Patient Image Uploader service requires in order to request this service. If you click the Type picker, you will see 5 default value types are shown. Once you add more than 1 type in here, the caller must send that value when request this service.

Set required arguments

If you look at Figure 348 and 354, you will notice the key values in the jsonPayload that we extracted from the DICOM inside the #region "Web API call Code" area. Those are acutally required from the POST method, i.e., locationId, patientId, dateTaken, and image data. So, the first step to define the Required Arguments is to define what we pass the arguments from the DICOM Reader to here. One thing we can consider is the value that's already extracted in the original DICOM Reader, e.g., patientId. Since the service uses the DicomFile.OpenAsync method to read the Dicom Tags, we don't have to use that method in the POST Patient Image Uploader service. In otherwords, all JsonContent related values need to be extracted in the DICOM Reader and be passed to here. The following Figure 361 shows each service's responsibilities.

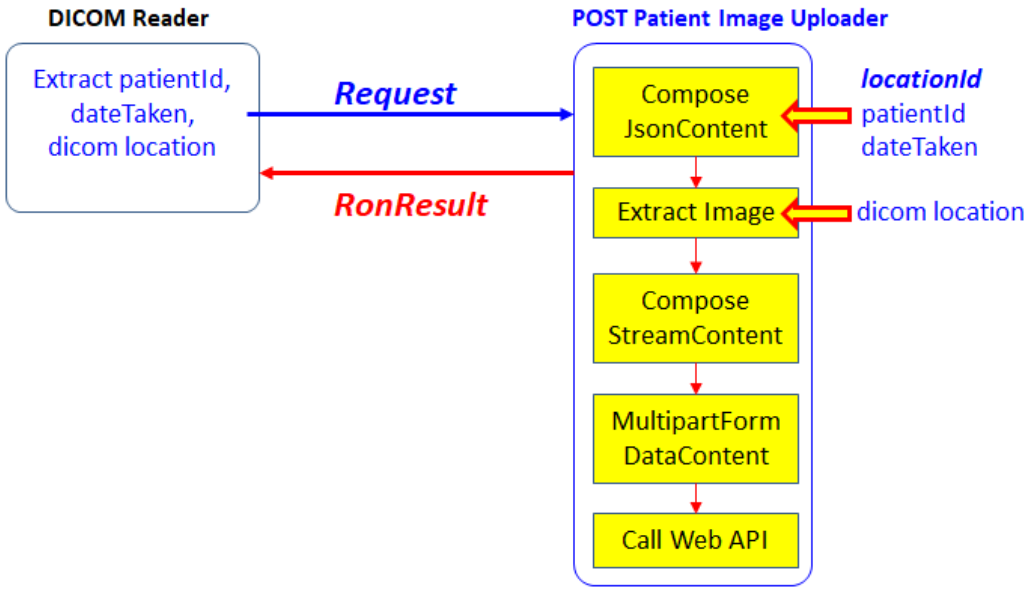


Figure 361.Request process between two services

Based on the worflow of the POST Patient Image Uploader service, we can easily identify what needs to be passed from the DICOM Reader. As a side note, the locationId is not necessary to pass because that can be retrieved from the Instance's Reference as _inst["SendingFacility"]. But you can pass if you prefer. So, we can define 3 arguments in here, as shown in Figure 362.

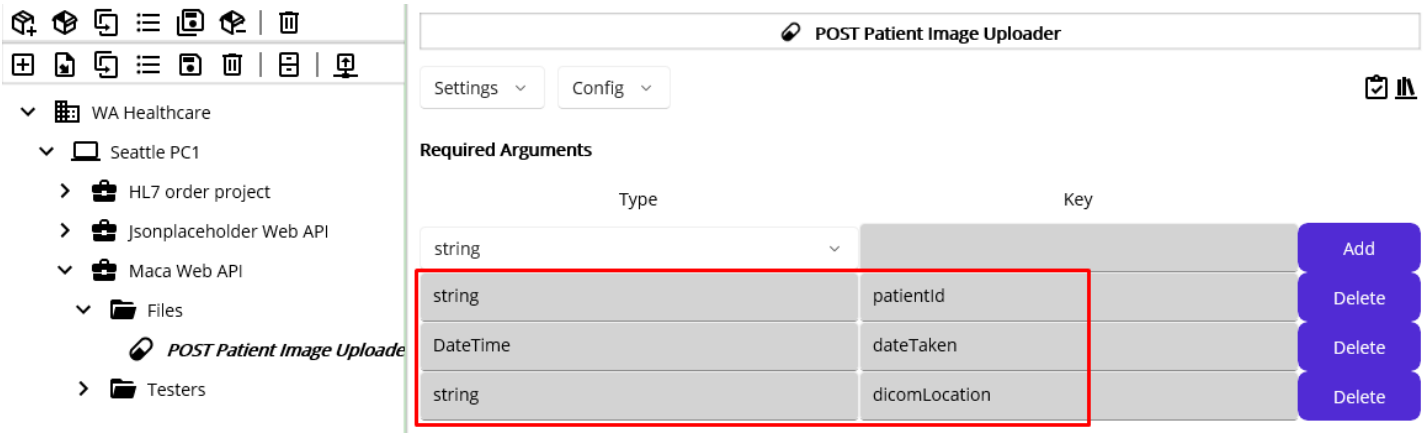


Figure 362.Required Argument of POST Patient Image Uploader service

Once you add the arguments, please save it so that the service is applied. Then these 3 arguments will be exposed to the calling service when it tries to call (request) the POST Patient Image Uploader. We will show that in a moment with examples.

Implement core logic

As we set how the service can be called (requested), the next step will be implementing the core logic with the passed parameters. In the previous steps, we inserted the Web API calling code inside the #region "Web API Call Code," as you see in Figure 354. So, go to the Execution > Steps view in the DICOM Reader Tester service and **copy that inserted code first**. Then come back here and go to Execution > Steps view. In the Code editor, **pase the code**, as shown in Figure 363.

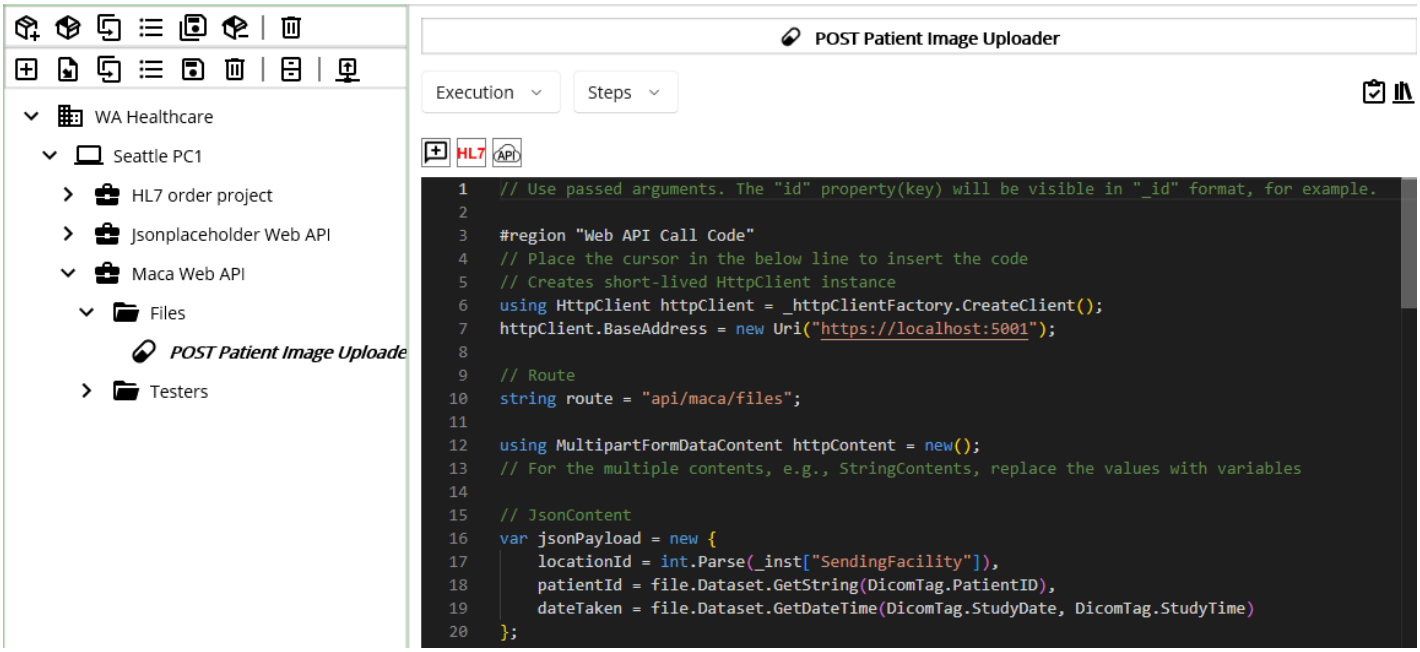


Figure 363.Pasted Web API Call Code

If you look at the top line, you will see the commented code that says the arguments that you defined can be used here in the form of underscore + name. Followed by the Figure 362, we can use the 3 arguments as _patientId, _dateTaken, and _dicomLocation.

Before we use that parameters, let's check the current code first to see if what happens with the copied code. Click the Verify Service ( ) button, and you will see an error message popup, as shown in Figure 364.

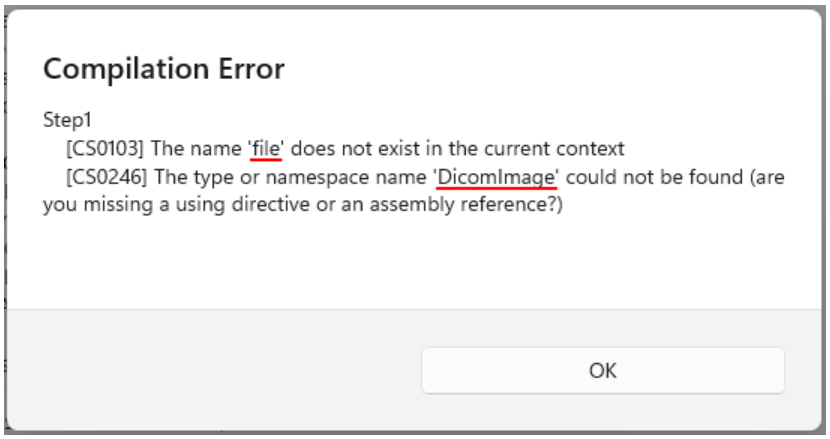


Figure 364.Undefined variable and missing namespace

The first issue related to 'file' can be resolved with the following code, as shown in Figure 365.

```
// JsonContent
var jsonPayload = new {
    locationId = int.Parse(_inst["SendingFacility"]),
    patientId = file.Dataset.GetString(DicomTag.PatientID),
    dateTaken = file.Dataset.GetDateTime(DicomTag.StudyDate, DicomTag.StudyTime)
};
```

```
var jsonPayload = new {
    locationId = int.Parse(_inst["SendingFacility"]),
    patientId = _patientId,
    dateTaken = _dateTaken
};
```

```
string streamContentName = "image";
string streamFileName = file.Dataset.GetString(DicomTag.PatientID) + ".jpeg";
httpContent.Add( streamContent, streamContentName, streamFileName);
```

```
string streamContentName = "image";
string streamFileName = _patientId + ".jpeg";
httpContent.Add( streamContent,
streamContentName, streamFileName);
```

Figure 365.Replace DICOM tag related values with the parameters

As you see, we used the parameters _patientId and _dateTaken to set the JsonContent's corresponding properties. This shows how the parameters can be used. We believe that this well describes the parameters, and let's keep checking the code.

The scond error is related to the **namespace** that we saw in the previous section. As we did in Figure 351, add following namespaces in the Using tab.

```
using FellowOakDicom;
using FellowOakDicom.Imaging;
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Formats.Jpeg;
using SixLabors.ImageSharp.PixelFormats;
```

Note. As we already added required DLLs in the previous sections, we will not go over the steps again. If you skip that section or freshly start over, click NuGet tab and search and download **fo-dicom.Imaging.ImageSharp** and **SixLabors.ImageSharp**, as we did in **Figure 349**. Then add DLLs as we did in **Figure 350**.

Once you add the namespaces, click the Verify Service button again. Then you will see the popup, as shown in Figure 366.

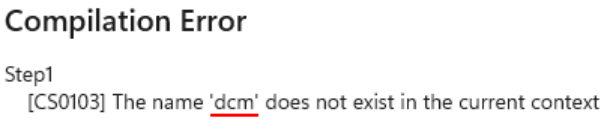


Figure 366,Undefined variable error

As you may notice, there is a _dicomLocation out of 3 parameters. And we can replace the value with the parameter, as shown in Figure 367.

```
// StreamContent
DicomImage dcmImage = new ( dcm );
IImage image = dcmImage.RenderImage();
```

```
// StreamContent
DicomImage dcmImage = new ( _dicomLocation );
IImage image = dcmImage.RenderImage();
```

Figure 367.Replace dcm with _dicomLocation

Now, you will see the success message popup, when you click the Verify Service button, as shown in Figure 368.

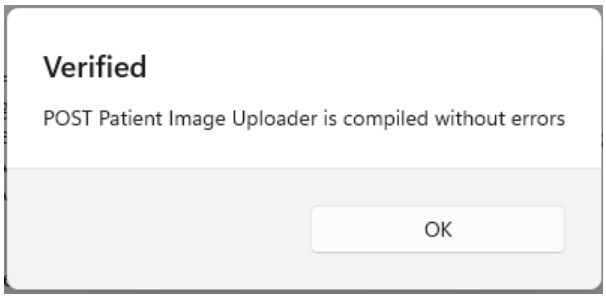


Figure 368.Successfully verified POST Patient Image Uploader service

Lastly, as we added route Reference in the Files group References area, we need to modify base address and route values that are hard-coded. Replace them with reference values like below code.

```
httpClient.BaseAddress = new Uri( _proj["baseAddress"] );

// Route
string route = _group["route"];
```

Although we didn't add any Filtering code or Deployment code, we can keep the service as-is to focus on the image uploading. So, do not add any further code and move on.

Test the service in the Management workstation

As we mentioned in the Benefits of code separation section, we can test this stand-alone service withtout using other services, i.e., test manually. Since this service executes the code once the request comes, we don't have to worry about any accidental code execution like CodeRepeater does when it's deployed. So, let's load and deploy this service.

To test this service, we logged in as **sidekick** account. Before load the service, **don't forget to deploy the Files group first**. Once you deploy the service, click the Request tab, as shown in Figure

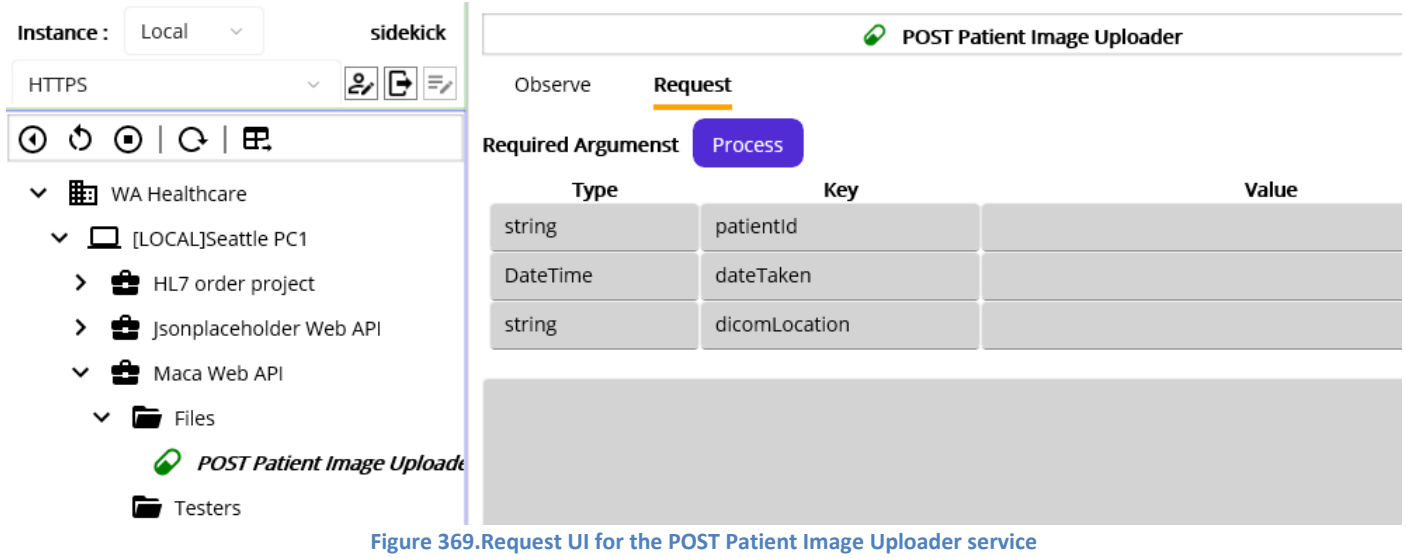


Figure 369.Request UI for the POST Patient Image Uploader service

At first look, you will notice that the UI is similar to the anonymous object input pane in the Web API client UI. The difference is it uses the entry area under the Value column. So, what we need to do here is type the value followed by the Type and Key. As we did before, we can reuse the value along with the actual file location, as shown in Figure 370.

Type	Key	Value
string	patientId	0001
DateTime	dateTaken	2025-04-10T14:19:14
string	dicomLocation	C:\Maca\Dicoms\CT0001.dcm

Figure 370.Sample test values for the manual request

Once you type the sample values, click the Process button. If you see the following error message below the input area, please make sure to run the Maca Web Api container first. This is an important point in terms of failed request because we may have to face a situation to reprocess the request manually with passed arguments, i.e., no manual typing in here. We will show this situation shorthly.

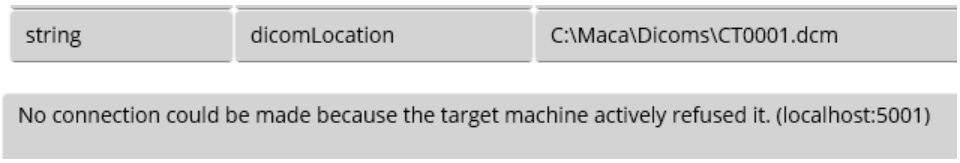


Figure 371.Error message when the target server is down

If the container is running, click the Process button again.

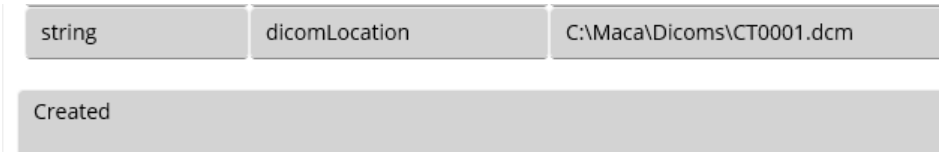


Figure 372.Successfully processed reuqest

As we saw the manual request was successful, we can check the transactions. Plase click the Observe tab. Then the results will look like, as shown in Figure 373.

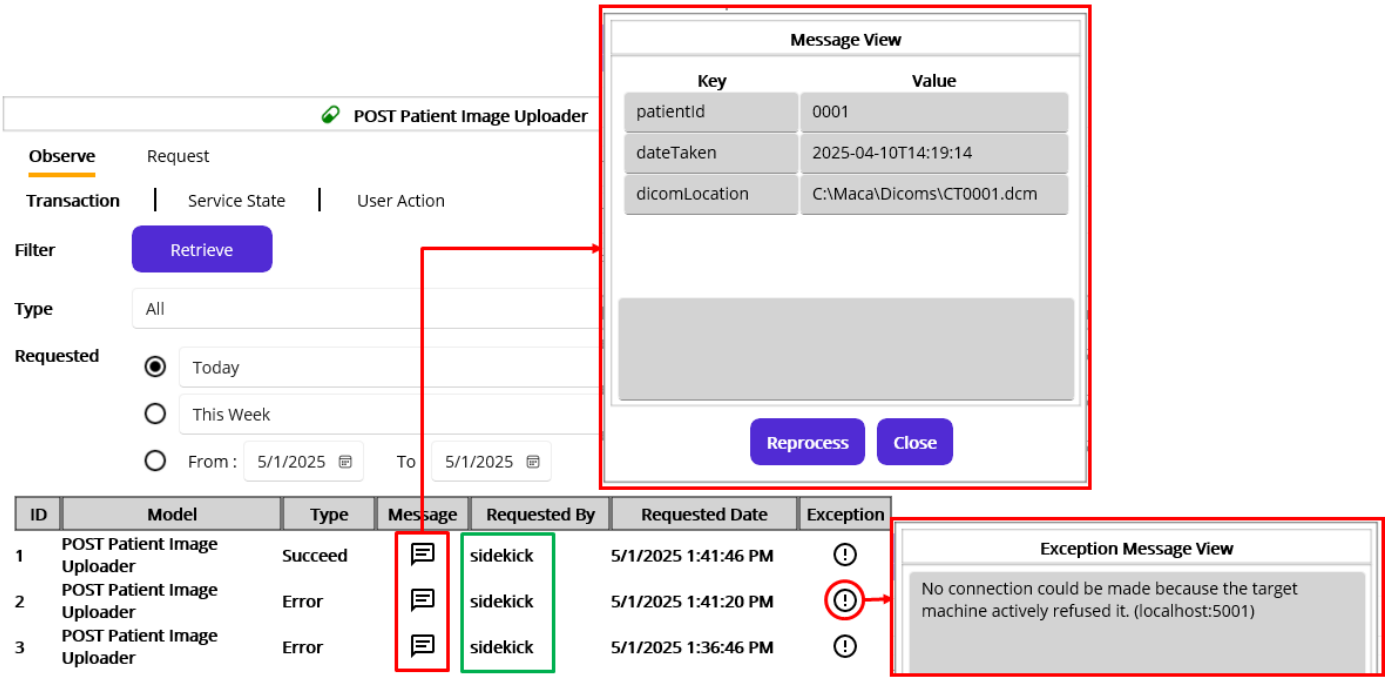


Figure 373.Manually requested transactions

As you see, the transactions were requested by the sidekick user. So, we can identify the transactions were manually requested. The second thing to note is the error message on the right. Since we didn't start the Maca Web API, we received an error message in the Request tab when we hit the Process button. And the message was saved as an Error in the database as a result. The last thing is the Message popup that shows the requested arguments at the top. As we typed the 3 test values, all values were saved in the database and be shown in the Message popup. Interesting thing in the popup is the Reporcess button that allows you to re-execute the service with the value listed on the popup. Based on the first 2 attempts, we couldn't submit the request because the server was down. Once we started up the container, the next request was successful.

Reprocess previously unprocessed requests

As we faced minutes ago, it would be possible that the RequestListener service can't process the request that contains arguments due to some unexpected reasons, such as the target server is down. That could be handled by keep trying till the request is processed or the Request Listener cacheing the request and process them later. But in case we need to reprocess previously processed requests, i.e., not the requests in the backlog or queue, we need a way to handle that case. We may use the Request tab by manually typing the values. But that's still not desirable due to some reasons, like typo. Instead, we can reprocess what was requested with the value saved in the database. That is the Reprocess button was designed as we mentioned in the HL7Client.

You can observe any RequestLisener services and open the Message popup to reprocess the request. To try this feature, shut down the Maca Web API container. This will make the same situation as we try to use the Request tab.

Click one of the message buttons to open the popup. Since we tried several times with the same arguments, any of the Messages will be OK to use for this test. But in real cases, you can pick the specific transaction from the results. When the popup shows up, click the Reprocess button, as shown in Figure 374.

Figure 374.Unsuccessful request due to the server down

As you see, we have the exactly same error message that we saw in the Request tab. Now, start up the Maca Web API container. When the server is up and running, click the Reprocess button again. At this time, you will see the Created message, as shown in Figure 375. And the server will show that 1 entity is created in the in-memory database.

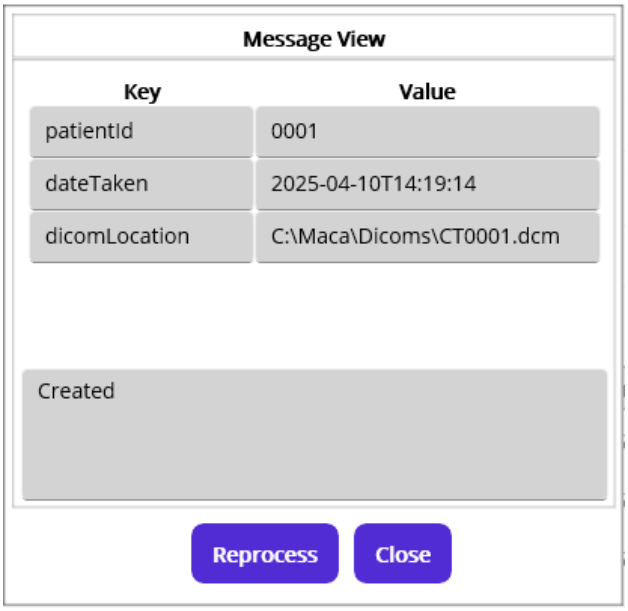


Figure 375.Successfully reprocessed request

Now, let's retrieve the service. Close the popup and click the Retrieve button. Then you will see 2 new entries at the top of the results, as shown in Figure 376.

ID	Model	Type	Message	Requested By	Requested Date	Exception
1	POST Patient Image Uploader	Succeed		sidekick	5/1/2025 2:30:57 PM	
2	POST Patient Image Uploader	Error		sidekick	5/1/2025 2:28:46 PM	
3	POST Patient Image Uploader	Succeed		sidekick	5/1/2025 1:41:46 PM	
4	POST Patient Image Uploader	Error		sidekick	5/1/2025 1:41:20 PM	
5	POST Patient Image Uploader	Error		sidekick	5/1/2025 1:36:46 PM	

Figure 376.Reprocessed requets transactions

By using the Reprocess feature in the transaction's message popup, you can manage old requests to be reprocessed. In addition, the requests will be tagged with the user account. So, the reprocessed transactions will be tracked by the requester's account.

So far, we practiced how the RequestListener service can be requested manually. Now, let's make the other service requests this serivce.

Request the POST Patient Image Uploader

Open the DICOM Reader Tester service and go to the Execution > Steps. In the Code editor, there is still exising old code that calls Maca Web API. **Please remove all code inside the region**, as shown in Figure 377.

```
// Sends a message to ORM Sender
// await RequestAsync(message, "8473612f-910b-4176-9ec0-bf0e435ec0da");

#region "Web API Call Code"

#endregion
}
```

Figure 377.Remove the Web API Call Code and place the cursor in it

Once you place the cursor where the red arrow points at, click the Add Service Request Code ( ) button to open Add Service Request popup, as shown in Figure 378.

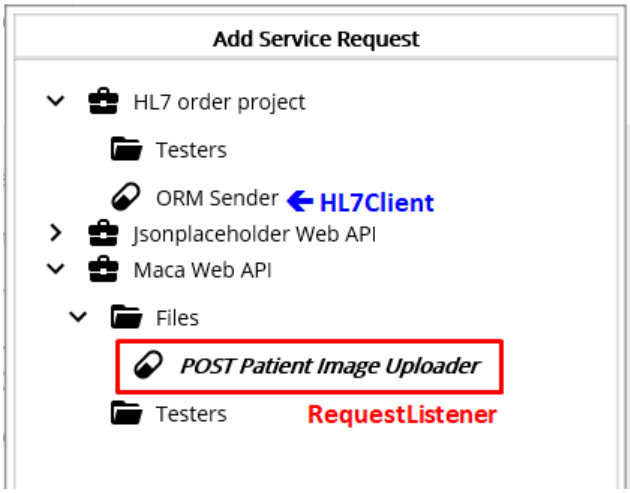


Figure 378.List of the requestable services

Previously, we could see only the ORM Sender, which is the HL7Client type. But at this time, the POST Patient Image Uploader is visible as well. This means the service is also callable, i.e., requestable. Select the service and click the Add button. And the auto-generated code that requests the service will be inserted, as shown in Figure 379.

```
#region "Web API Call Code"
// Please assign required argument's value to send to POST Patient Image Uploader
var requiredArgs = new JsonObject()
{
    ["patientId"] = , //assign string type
    ["dateTaken"] = , //assign DateTime type
    ["dicomLocation"] =  //assign string type
};
RonResult reqResult = await RequestRonResultAsync( requiredArgs,"da48ef94-baed-4c1a-a9ec-4654f8eb774a" );

#endregion
}
```

Figure 379.Inserted service request code

If you look at the red box, you will see the 3 arguemtns that we defined in the POST Patient Image Uploader service. Those are commented with the type that each argument requires. Followed by the direction, it is required to assign a value with right type. Based on the previous code, we can set the properties like below Figure 380

```
#region "Web API Call Code"
// Please assign required argument's value to send to POST Patient Image Uploader
var requiredArgs = new JsonObject()
{
   ["patientId"] = file.Dataset.GetString(DicomTag.PatientID), //assign string type
   ["dateTaken"] = file.Dataset.GetDateTime(DicomTag.StudyDate, DicomTag.StudyTime), //assign DateTime type
   ["dicomLocation"] = dcm //assign string type
};
RonResult reqResult = await RequestRonResultAsync( requiredArgs,"da48ef94-baed-4c1a-a9ec-4654f8eb774a" );

result.TransactionResult = reqResult.TransactionResult;
result.ResultMessage = reqResult.ResultMessage;
#endregion
```

Figure 380.Request POST Patient Image Uploader service

Since we used the populating code before, there will be no difficult code to understand. One thing to note is the last two lines that add the result of request to the RonResult. Since the result is pre-defined in the code, we set the returned RonResult's variable as "reqResult" instead. With this setup, the transaction will also track the result of the request. Once you modified, click the Verify Service button to see any issue comes up. If there is no issue, please load the service. FYI, we still commented the ORM Sender calling code. Switch to the Management workstation, and there will be two services, as shown in Figure 381.

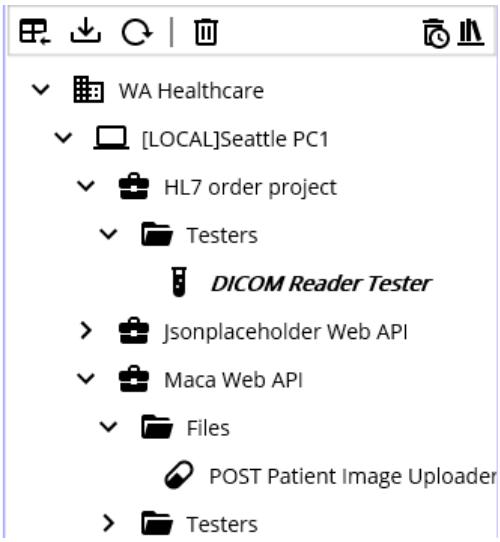


Figure 381.Two associated services

Before we deploy services, please shut-down the container again. Since we will use the existing DICOM file, we may have the repeating 409 conflict errors. To avoid that situation, restart the container. So, we don't have any patient image on the server and can upload using both services.

To make the service works properly, please deploy the POST Patient Image Uploader service first if that's not deployed. Then deploy the DICOM Reader Tester service. Once the deplyment is done, retrieve DICOM Reader Tester, as shown In Figure 382.

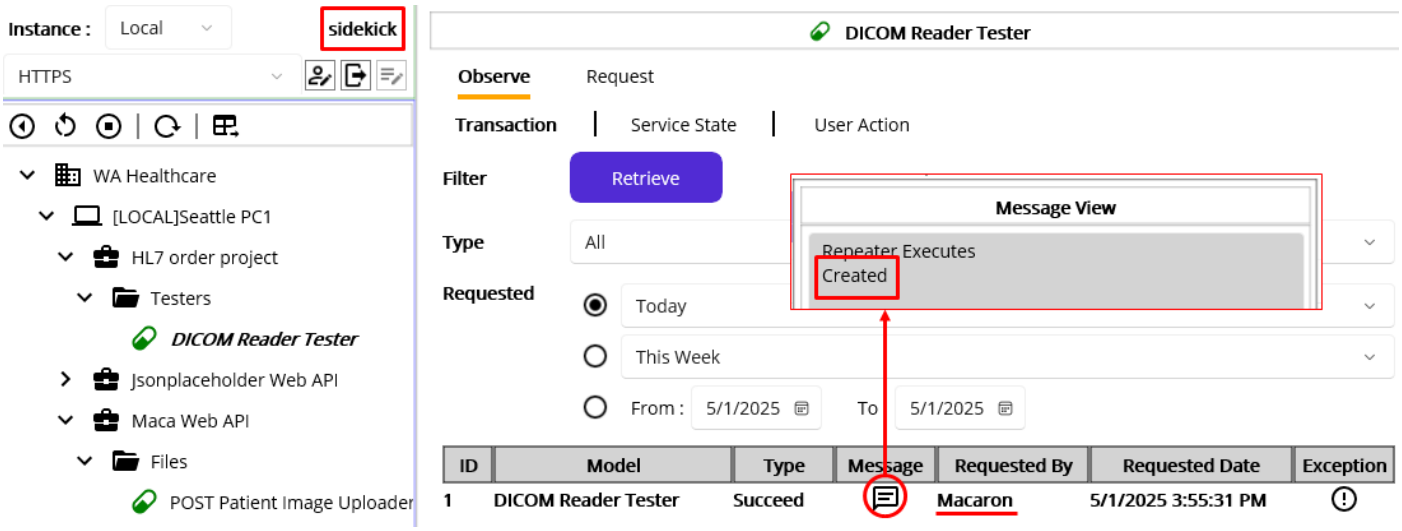


Figure 382.Success message in the DICOM Reader Tester service

As you see, the service's last transaction has the result of the request, and the popup shows that in the Message. Please note that the transaction was not generated by the sidekick user. So the Requested By is marked as Macaron. Now, click the POST Patient Image Uploader service and click the Retrieve button, as shown in Figure 383.

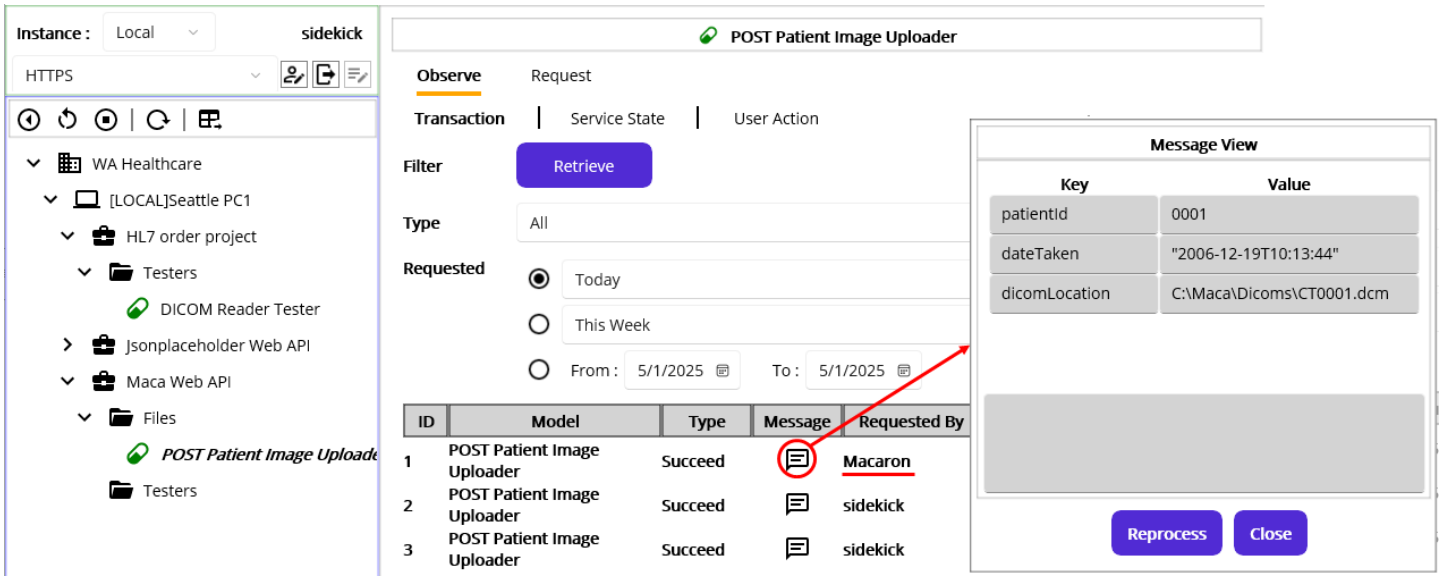


Figure 383.POST Patient Image Uploader service transactions

At this time, you will see the familiar result that we saw before. Just the Request By is different that shows the last transaction was not requested by the sidekick user. Nothing special here, so we can move on to the next step.

Apply the new code in the working service

As we practiced and tested, we could separate the Web API calling code and made it as a requestable service. So, the POST Patient Image Uploader service could be production level service.

Note. We have focused on the core requirement implementation only. All other points, such as error handling and requests management, are not included here. Any unstated or missing parts related to the hypothetical requirements and the services you may see. So, please let us know anything that can improve the functionality of MACA suite.

And the last thing is the DICOM Reader service. We tested the new feature by cloning the service. Based on what we modified in the cloned service, all we can do with the working DICOM Reader is insert “Web API call Code” area at the end of the code in the Execution > Steps Code editor. To achieve that, we can choose one of the following approaches.

1. Use cloned service to re-clone(new GUID will be generated) and replace existing working DICOM Reader
2. Insert the code in the working DICOM Reader.

You can choose one of them, but please make sure to have a way to track its history because Maca doesn't provide a version controlling feature. In the later version, we may have a Git associated feature, but not in near future.

So, finalize the requirement implementation based on your decision.

Implementing patient PDF download

The WA Healthcare IT team shared the detail that it needs to keep downloading (requesting) the GET method every 1 minute. To make sure the calling location is not confused, the location id must be provided at the end of the request. As we know, the location id 1 is Seattle and 2 is Bellevue. We are working on Seattle location, so we will use 1 for the location id. And the response data will be a zip format file containing at least 1 PDF file. Since it requests every minute, there would be no file to download. So, please make sure the downloading service won’t mark that case as an error. But if there is, the service extracts the downloaded zip file to the following folder.

C:\Maca\Temp\Results

As you may consider, the requirement can be implemented with the GET ID style of the GET method. If you look at the spec in Figure 384, you will notice that the locationId comes after the route, which is exactly the same as what we practiced with the Jsonplaceholder API’s GET.

Figure 384.GET Patient PDF spec

As a first step, create a CodeRepeater service named GET Patient PDF Downloader under the Files group, as shown in Figure 385.

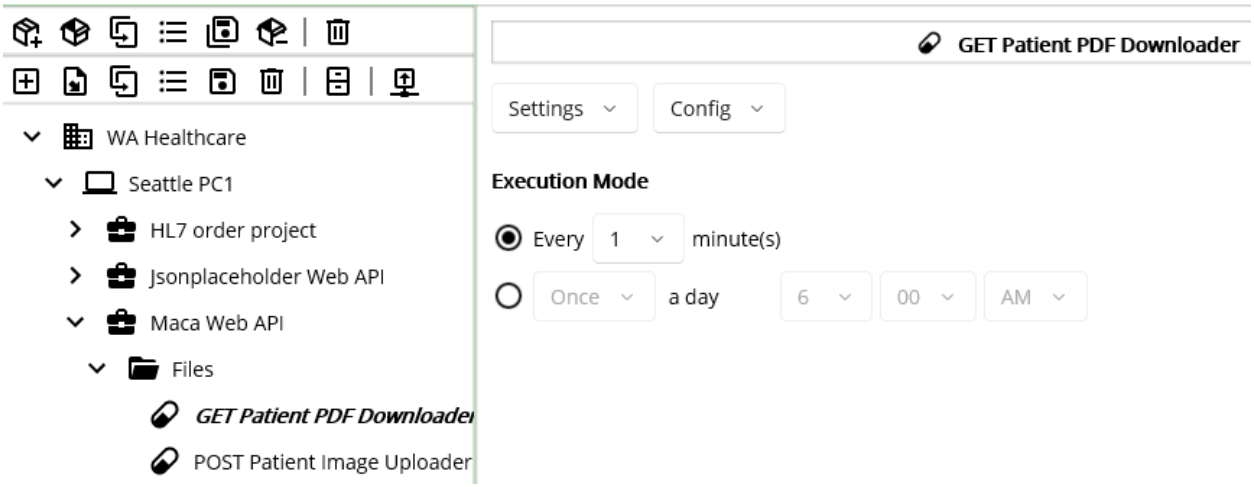


Figure 385.GET Patient PDF Downloader service

Then go to Execution > Steps view and click the Web API ( ) button. As we added route Reference, we use base address from the project and route from the group, as shown in Figure 386.

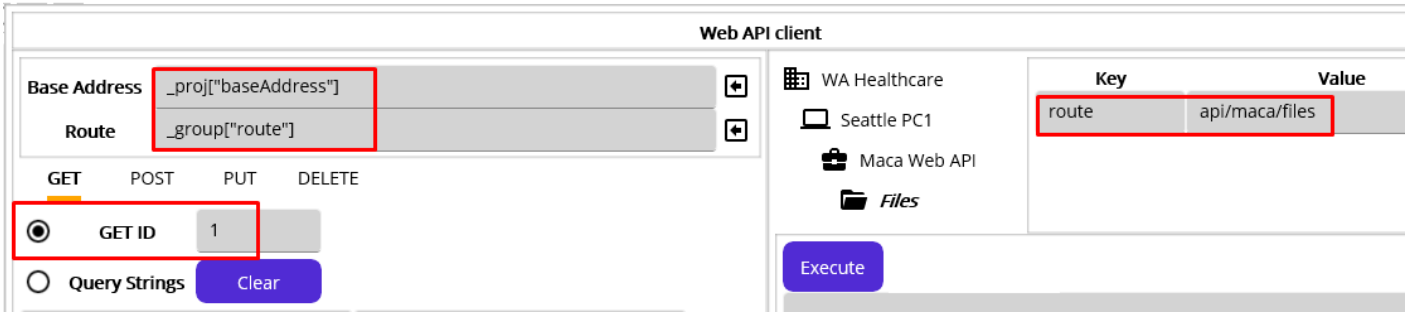


Figure 386.Basic GET template

Type value 1 in the GET ID area and click “Insert code and close” button.

Figure 387.Keep the required code only

As you see in Figure 387, remoe the rest and keep the code that we need.

Modify the location id

As we did in uploading steps, we can use the location id to reference from the instance model, as shown in

```
// GET ID: Replace the value with a variable, e.g., /{_id}
routeQuery.Append($"/{_inst["SendingFacility"]}");
```

Figure 388.Add instance reference SendingFacility

Handle non 200 cases

Followed by the direction, we need to handle 204 case that returns no data. In that case we can set the mressage and return. That will mark the transaction as success but with a message “No file to download.” If we have non 200 status code as a result, then mark the transaction as Error and stop the rest of the code. And the error message will be saved in the database. The following Figure 389 handles those cases.

```
int statusCode = (int)response.StatusCode;

// status code 204 NoContent
if ( statusCode == 204 )
{
   result.ResultMessage = "No file to download";
   return result;
}

if( statusCode != 200 )
{
   result.TransactionResult = TransactionResults.Error;
   result.ResultMessage = await response.Content.ReadAsStringAsync();
   return result;
}
```

Figure 389.Handles non 200 cases

Extract the downloaded zip file

Once there is no issue, we need to extract the downloaded zip data to the pre-defined folder.

```
try
{
   using Stream stream = await response.Content.ReadAsStreamAsync();

   // fileDir can be added in the group References
   string fileDir = $"C:\\Maca\\Temp\\Results";
   Directory.CreateDirectory(fileDir);

   StringBuilder buffer = new();
   using (ZipArchive archive = new ZipArchive(stream, ZipArchiveMode.Read))
   {
      foreach (var entry in archive.Entries)
      {
         string filePath = Path.Combine(fileDir, entry.FullName);
         entry.ExtractToFile(filePath, overwrite: true);
         buffer.AppendLine($"Extracted: {filePath}");
      }
   }

   result.ResultMessage = buffer.ToString();
}
catch (Exception ex)
{
```

```
    result.TransactionResult= TransactionResults.Error;
    result.ResultMessage = ex.Message;
}
```

Figure 390.Extracting downloaded zip data

As we did with the base address and route, you may put the extracting directory to the project or group References. Please do that if that works better for your environment.

Then the code reads Stream data from the response and parses it using ZipArchive class. Since the class is under the System.IO.Compression, we have to add that namespace in the Using tab in order to avoid compilation error. And the System.Text namespace is also required due to the StringBuilder. Figure 391 shows added namespaces.

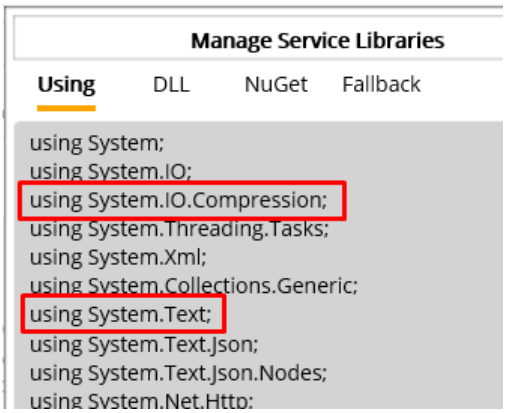


Figure 391.Required namespaces for ZipArchive and StringBuider

Note. StringBuilder is one of the common classes that can be used in the Code editor. It might be cumbersome, but to encourage a user practicing the Using tab, we did not set System.Text inside the pre-declared namespaces on purpose.

Once the ZipArchive is open, each internal file will be saved to the target directory. If the same file exists, that will be overwritten with a new one.

Lastly, if there is no issue, then the downloaded files list data would be saved to the database.

That's all of the main logic. Although the details of each class and method are not mentioned, the overall process will be easily followed.

Now, click the Verify Service (📋) button to see if any error comes up. If not, load the service. Then switch to the Management workstation.

Check the download folder

If you did not shut down the container, there would be 1 uploded image under the location id 1. Before deploying the downlader service, let's check the target folder first, as shown in Figure 392.

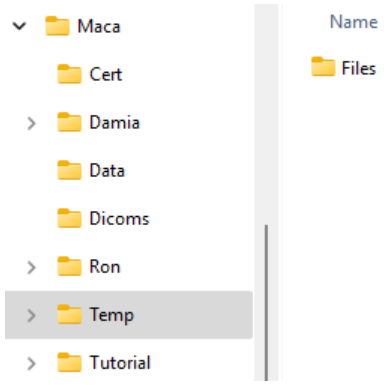


Figure 392.The taget destination to download PDF

Although the folder is not the final folder to extract, we can look at the folder to verify the result of the 1st execution of the downloader service. We are now ready to run, so telect the GET Patient DPF Downloader service and deploy.

Verify the result of the 1st execution

Click the Instance tab and retrieve the downloader service's transaction, as shown in Figure 393.

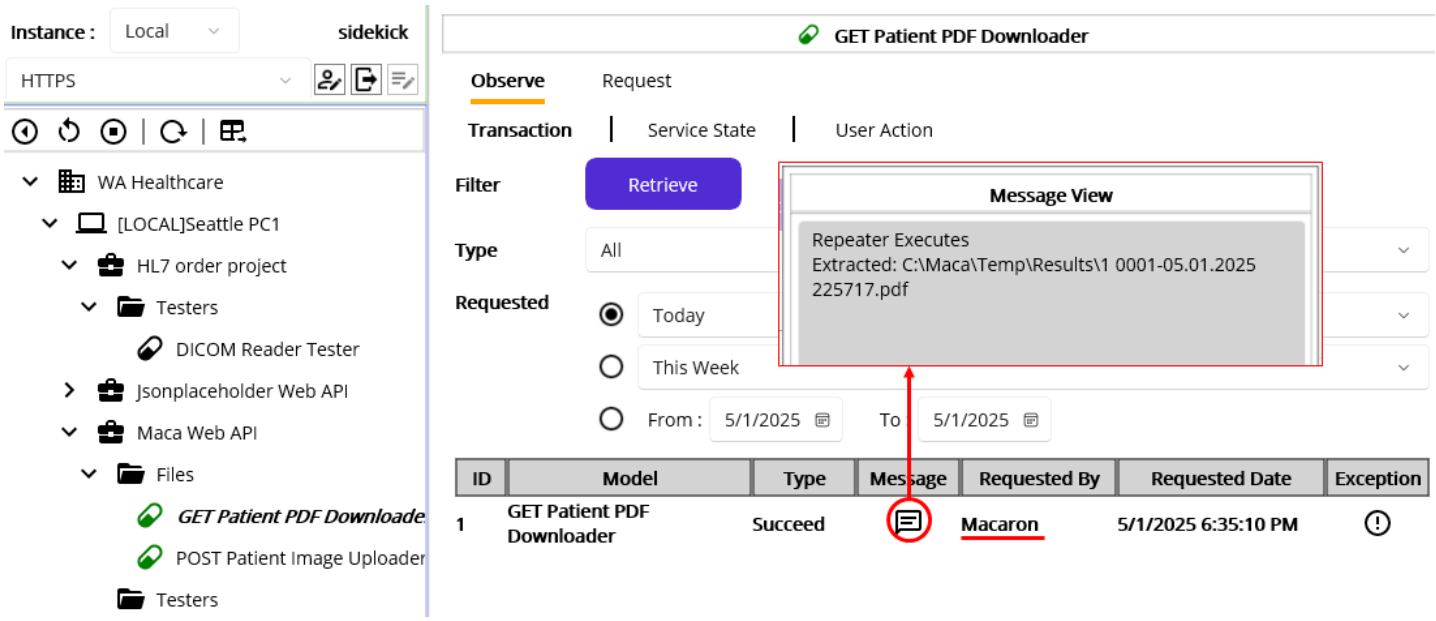


Figure 393.Succeed 1st transaction

If you click the Message, you will see the saved file name along with the saved location. Go to the location, and you will see the downloaded pdf file and its content, as shown in Figure 394.

Figure 394.Downloaded pdf from the server.

Although we uploaded 1 image, you can upload as many images as possible as long as the patient id is different. But if multiple files are uploaded within a minute, e.g., 10 patients' images, not all pdf files will be downloaded even if the download cycle is 1 minute. Max 3 PDF files in a zip file per request will be downloaded by design. And the PDF generation process will run every 2 minutes. You can try this by either uploading images fast or changing the download interval to 10 minutes. But be aware that it uses in-memory database, and you may have a storage issue if you keep uploading huge images.

Check the no file to download case

While we checking the download files, the GET Patient PDF Downloader service kept running. So, let's retrieve the service again, as shown in Figure 395.

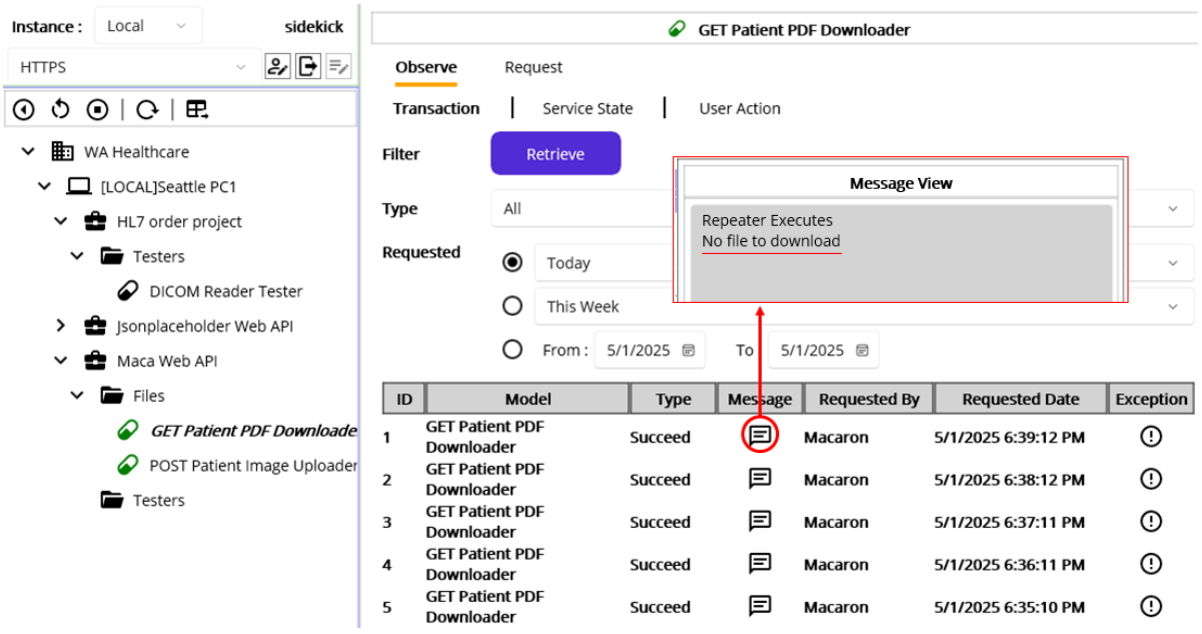


Figure 395.Transactions with no file to download

As you see, there is no Error transaction in case of 204 No Content. So, the requirement was implemented correctly.

You can modify the code to check what the result is when the location is 10, i.e., no such id in the server. Or, shut down the Maca Web API and request. Again, we are focusing on the core logic only, so you can test or verify the error cases on your own.

Wrap-up

It was a bit lengthy steps to implement new requirements compare to the sceenario 2. In order to build the services, we spent huge amount of time in practicing the Web API client UI along with the Jsonplaceholder API and the Maca Web API docker image. Although the actual implementation didn't cover test cases thoroughly, previous test experience through scenraio 1 and 2 will give you an idea of what needs to be tested. In addition, the way we structured the Web API project will be another good experience that can be applicable in your own project later. Please take advantage of the Maca's UI and its features in your project implementation.