Inhalt

[Introduction 1](#_Toc504050763)

[Architecture Overview 2](#_Toc504050764)

[Loops 2](#_Toc504050765)

[MHL – Message Handling Loop 2](#_Toc504050766)

[EHL – Event Handling Loop 2](#_Toc504050767)

[UI Loop 2](#_Toc504050768)

[Error Loop 2](#_Toc504050769)

[Logging Loop 3](#_Toc504050770)

[Timing Loop 3](#_Toc504050771)

[Dialog Loop 3](#_Toc504050772)

[WebService Loop 3](#_Toc504050773)

[Oracle Loop 4](#_Toc504050774)

[INI File (ServerINI.txt) 4](#_Toc504050775)

[Webservice (WS) (DashboardService) 4](#_Toc504050776)

[Publish Webservice 4](#_Toc504050777)

[Webservice Methods 5](#_Toc504050778)

[Architecture Details 5](#_Toc504050779)

[Common Code for all Loops 5](#_Toc504050780)

[References Cluster 5](#_Toc504050781)

[Data Cluster 5](#_Toc504050782)

[Common Messages 5](#_Toc504050783)

[ErrorHandler.vi 6](#_Toc504050784)

[Initialization (Init.vi) 6](#_Toc504050785)

[Boradcast Data (SendToAll.vi) 6](#_Toc504050786)

[Queue Tracker (Debug\_QueueTracker.vi) 7](#_Toc504050787)

[Building the EXE 7](#_Toc504050788)

[Include the Webservice 7](#_Toc504050789)

[Include “Always Included” VIs 7](#_Toc504050790)

[Include the INI-File 7](#_Toc504050791)

# 

# Introduction

Welcome to the CER Dashboard Server-Application. This document is intended to give future developers (and maybe previous who forgot everything) an overview of the architecture and the functionality of this Application.

The Server has the following Tasks:

* Query the Oracle Database for all CER Service Requests
* Publish these Service Requests to all Client-Applications on the AE PCs
* Track which clients are currently connected

The Mechanisms to achieve these Tasks are explained throughout this document.

# Architecture Overview

The Server Application uses a Queued Message Handler (QMH) Architecture. It was build from scratch and not from the QMH Template Project. Though the MessageQueues.lvlib is reused here but modified in some ways.

The Application consists of this QMH and a Webservice. The Webservice is the Interface for the Client to communicate with the Server. QMH and Webservice communicate with each other using an FGV.

## Loops

### MHL – Message Handling Loop

The MHL has less functionality than in the basic Core 3 Template QMH. In this Framework it is exclusively responsible for forwarding (handling) messages. There is no logic implemented and there are no Frontpanel Indicators within this Loop, the UI Loop is responsible for updating them. It just receives Messages and forwards it to the Loops the Message belongs to. The functionality is quite simple and its easy to see the Messaging Structure within here.

### EHL – Event Handling Loop

The EHL is just as in the normal QMH Template. It just receives User Inputs from ALL Frontpanel Controls and forwards their data to the MHL.

There is a User-Event here used to send an Exit-Message just as if you press the Stop button. We will refer to this later in this document. The User Event will be used in case of error or other reasons for programmatic shutdown of the application.

### UI Loop

The UI Loop is responsible for updating all Frontpanel Indicators. It contains one case for each indicator which preferably is named just as the Indicator name or similar. So if I want to Update “CurrentSRs”-List, I send a message with the correct Datatype for this to the UI Queue and it will be displayed in the indicator.

This is different to the QMH Template in which the MHL is responsible for that. But here you need local variables to update an indicator from different MHL-Cases. With the UI Loop all Cases of the MHL can enqueue Messages and can stay aligned to the QMH-Structure (we don’t want Locals/Globals!).

### Error Loop

The Error Loop is quite simple and shouldn’t need big changes later on. It simply waits for “Error”-Messages. If one occurs, it will check if the error is critical (you can define errors that shall be ignored) and if it is, it will forward the error to the Logging-Loop to log it to a file, and then fire the User Event mentioned in the EHL-Section to stop the Application. The User Event will contain the error, and the EHL will receive it, send the Exit Message (everything will stop) and then output the error to the final simple error handler on the far right.

The Error-Messages in the Error Queue come from the Errorhandlers placed at the end of every Loop. I will refer to their exact functionality later, but they will send every error/warning to the MHL Queue which will forward the Error-Message to the Error Queue. That way all Errors within the whole Application will end up in the Error Loop and can be filtered, logged and evaluated if they are critical so that the Application should be shut down.

Its possible to add code here that instead of shutting down the application will react to some errors by sending messages to the MHL.

### Logging Loop

The Logging Loop will be Initialized in the Beginning of the Application and open a TDMS-File Reference here. It stores this reference, and then receives Messages with Data to be logged. A Log contains a Type (e.g. Notification, Error, User Acitvity …), a LogText (e.g. Error formatted into string) and a Timestamp. These will be logged into different Channels within the Logfile. Also, each day will get its own channelgroup. The Path for the Log-File is stored in the ServerINI.txt in the data-folder.

### Timing Loop

This Loop sends cyclic messages to the MHL. It can be used to time other Loops. If an operation (such as Reading the currently connected Users should be done frequently you could implement this within the responsible loop, but in the ideal case all Loops should only react to commands. This Loop is a central point to time all cyclic operations within the Framework. Its currently only used for reading the Client Users, but can be easily extended to send many more messages.

### Dialog Loop

This Loop is a generic Dialog Launcher. It will launch asynchronous VIs (call and forget). These VIs have to show their Frontpanel when loaded and need to close themselves. They also need to contain an Errorhandler. All Dialogs MUST have the same interface (necessary for calling them in asynchronous call node). Their Inputs are the References Cluster which contains all Queues of the Main.vi, a Variant that can contain custom Data that needs to be transferred to the Dialog and an Error Input. With the References Cluster the Dialogs can write directly to the MHL or to other Queues (Other queues are not recommended since its against the QMH Architecture).

Currently there are two dialogs implemented:

* TestClient.vi  
  This VI will send simple HTTP-Requests to the Webservice to see if it is running properly
* Settings.vi  
  This Dialog contains a Control of type ServerINI.ctl. It can be used to update the INI-Data while the Application is running. You can choose if you want to update only the current runtime-Data or if you want to overwrite the Data stored in the INI-File as well.

### WebService Loop

This Loop is responsible to communicate with the Webservice.

The Project contains a Webservice that has currently 2 Methods. One method is called periodically by all clients (Heartbeat.vi) and will write the current Client-Name and a Timestamp to the WebServiceFGV.vi. The other Method is GetSRs.vi and it will read the currently fetched Service Requests from the WebServiceFGV.vi. The Methods are quite simple, details on the webservice will be mentioned later.

The Webservice Loop will simply read and write the WebServiceFGV.vi and update the new SRs in it, and check how long Users haven’t send a heartbeat. If the user didn’t send a heartbeat within the last few seconds (defined by Client Timeout in INI file) he will be deleted from the Client List and is not seen as connected anymore.

### Oracle Loop

This Loop contains Functions of the Database Connectivity Toolkit and is responsible for reading the Service Requests from the Oracle Database. It therefore opens a connection on Initialization-Message and then constantly reads the Database and sends the most recent data to the MHL.

Reading Oracle takes a while with the current configuration (30-60 seconds). This is a problem for the reactivity of the Application. If you hit exit, this loop might be stuck in a SQL-Query. In this case all Loops will stop, and a Dialog will appear telling the User that Oracle needs to finish its query. Sadly the Oracle Query can’t be cancelled. We tried launching it via VI Server in a SubVI and then simply abort the SubVI. But this resulted in a Window telling “Resetting VI …” which took just as long as the query itself. So this is something we can’t really work around.

## INI File (ServerINI.txt)

The Application uses an INI-File (ServerINI.txt). It stores a Cluster (ServerINI.ctl) flattened to XML. The Path to this File is stored in INIpath.vi. This VI outputs “<application directory>/data/ServerINI.txt”

The INI File should be deployed together with the EXE. If the Application starts and no INI-File is found, the INI-File contains the wrong data or is empty, a Dialog (INIdialog.vi) will appear and asks the user to enter the necessary data manually. When this is done, the INI-File is created, and the entered Data is passed to the application for a normal start.

Later the User can use the Settings-Dialog (Settings.vi) to change the INI Data. He can do this just for the current execution, or he can choose to overwrite the INI-File.

## Webservice (WS) (DashboardService)

The Webserver is responsible for publishing all Data in the Network. All clients will use HTTP-Requests to call the Methods of the Webservice and send/receive Data.

### Publish Webservice

Since this webservice uses an FGV to communicate to the QMH, it CAN’T BE PUBLISHED! Publishing it would result in the Webservice running in its own Application instance (context). So while Development the QMH would run in context of LabVIEW.exe (Development System) and the WS would run in the context of the Webservice-EXE (don’t know the name right now), which is a different application. So each would have its own instance of the FGV and no Data would be transferred!

There are 2 options of making the Webservice accessable. Read these two subpoints, or see these Links for more Information:

<http://zone.ni.com/reference/en-XX/help/371361P-01/lvconcepts/ws_web_server/>

<http://zone.ni.com/reference/en-XX/help/371361P-01/lvconcepts/ws_distributing/>

#### Development:

Simply Start the Webservice. It will then run in “debug mode” (as LabVIEW calls it) on Port 8001. Keep in Mind, that this mode is very slow and the methods might even be completely blocked if some dialogs in LabVIEW are open.

#### Deployment:

When you build the exe, include the Webservice and assign a port (e.g. 8002) for it. After building the exe, it will contain the Webservice and host it within the Application context of the built exe. This way of publishing gives you full performance of the Method VIs and they still run in the same context as the QMH. This is the way to run the application on final deployment.

### Webservice Methods

#### Heartbeat.vi

This Method is called frequently by all Clients. It notifies the Server, that the client is still running properly. This way the Server can determine how many Clients are connected currently and for how long they have been connected.

The Client sends one Parameter (ClientName) to the Webservice. A Timestamp is taken within the Method each time a heartbeat arrives. The response of this Method is not used by the clients.

All code here (taking timestamp, adding client to list …) is implemented within the WebServiceFGV.vi (CMD=”Heartbeat”).

#### GetSRs.vi

This Method has no input-parameters and simply responds with the current List of Service Requests available on the Server. This Methods Output-type is set to Stream in the Webservice Properties. It will flatten the “SR List.ctl” to XML and use the “Write Response.vi” from the webservice palette to transmit exactly this string as HTTP-Body.

The Method acquires the SR List from the WebServiceFGV.vi which is updated with the latest Data in the Webservice Loop.

# Architecture Details

## Common Code for all Loops

### References Cluster

Every Loop will Receive the References Cluster (CommunicationReferences.ctl). This cluster stores all Queue References and the User Event. Usually loops should only obtain their own queue and the MHL from this cluster. Sending messages directly between the loops without using the MHL might lead to unstructured Messaging within the Framework and can result in a lot of debugging work. (I have experienced this already, be careful with this!)

### Data Cluster

Most Loops will contain a Data Cluster (e.g. ErrorLoopData.ctl). This contains mainly the INI-Data the loop needs. For the logger-loop this would be the file-path to the log-file. A loop won’t necessarily need this Cluster, but its recommended if you need to receive/store data from the INI-File

### Common Messages

All loops must contain Cases for these Messages

#### Initialize

This Message is sent on Application Startup and contains the whole ServerINI.ctl Cluster. This way all loops get the INI data and can pick the data they need from it. The loops will then store this ini-data in their own data-cluster.

#### UpdateSettings

This Message is similar to Initialize, in many loops it’s the same code. It is sent when the User uses the Settings.vi Dialog to update the INI-Data. This will result in the UpdateSettings Message and all loops will get the new data, pick what they need from it, and update these values in their data-cluster.

The difference to Initialize is, that the loops might close and reopen references here. E.g. the logging loop will close the current log-file and then reopen the logfile with the new data-path contained in the Message Data.

So this Message basically combines an Exit and an Initialize Message, but without stopping the loop.

#### Exit

The Exit message will stop the loop (there is a true constant wired to the loop condition in this case) and might have the loop close references.

#### <Default>

The Default case will catch all Messages that are not handled (just as in QMH Template). This case will generate an error telling which message was received in which loop.

### ErrorHandler.vi

Every Loop MUST contain the ErrorHandler.vi. This VI is Reentrant to avoid coupling between the loops. Its basically just a wrapper for the ErrorHandlerFGV.vi. This FGV is Non-Reentrant. It will only be called if an error occurs within a loop.

The FGV will get the reference to the MHL in the Init.vi and stores it throughout the execution. Whenever an error occurs its will be send to the MHL-Queue and then deleted. The output of the ErrorHandler.vi will only output an error, if the MHL Queue has been destroyed. In this case proper Errorhandling is not possible anymore, and the loop will stop directly. In all other cases the Error will first be evaluated by the error loop and will propably result in a clean Exit-Message (all loops will close all references and so on).

C:\Users\jgoebel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\ErrorHandler.png

## Initialization (Init.vi)

This VI reads the INI, Initializes the Queue Tracker (see Queue Tracker for more details), creates all Message Queues and the User-Event, Initializes the ErroHandlerFGV.vi and sends the Initialize-Message to the MHL. The MHL will then forward the Initialize-Message to all Loops with the ServerINI.ctl Cluster as Message Data.

At the end of the Init.vi the ErrorHandler.vi is called to make sure, that Initialization Errors will be sent to the Error Loop and handled properly.

## Boradcast Data (SendToAll.vi)

This VI is used to send Messages to all Queues. It’s currently used for the Exit, Update Settings and Initialize Messages.

If a loop is added, another EnqueueMessage.vi should be added to this VI.

## Queue Tracker (Debug\_QueueTracker.vi)

This VI is placed in the Enqueue Message.vi. It will open the Reference to a new Queue (“TrackerQueue”) and write every message that is enqueued to any of the Queues in the Framework to this Tracker Queue. It’s a lossy enqueue and the Queue Size is limited to 100 Elements to avoid a memory leak.

The Tracker Queue can be read by the “DebugCommunication.vi” in the Message Queue.lvlib. This VI will present all sent Queue Messages within the QMH with a lot of additional Data. (Message Data, Timestamp, Callchain of VI that sent the Message, Queue it was written to)

This is useful to track the Messages that are sent throughout the QMH. Since the QueueTracker is an FGV (it will open the Queue Reference to TrackerQueue in the Init.vi and close it in the DestroyCommunication.vi), it is non-reentrant. This will bring coupling to all the loops on a level that might affect performance dramatically (the Enqueue Message.vi is sent called very often). For that reason the VI is placed in a conditional Disable Structure. The Conditional Disable Symbol (“DebugMode” = True/False) is set in the Dashboard.lvproj Projectfile. For final tests you should disable this symbol and then see how the framework behaves.

More infos on Conditional Disable Symbols:

<http://zone.ni.com/reference/en-XX/help/371361H-01/lvhowto/creating_cond_disable_struc/>

<http://zone.ni.com/reference/en-XX/help/371361H-01/lvhowto/creating_custom_symbols/>

# Building the EXE

When building the EXE there are basically 3 things to keep in mind:

## Include the Webservice

You must include the Webservice in the Webservices-Section of the Buildspecification. Simply set the checkmark and define a port the Webservice should run on. Make sure NOT TO START THE EXE FROM A NETWORK DRIVE! This will result in the Webservice not being hosted without notification.

## Include “Always Included” VIs

Some Dialogs are called asynchronously (in the DialogLoop.vi). For all those VIs there is a Bookmark (e.g. “#AlwaysIncluded Settings.vi must be always included”). Find all those VIs with the Bookmark Manager before building the exe and add them to the always included section.

## Include the INI-File

The INI-File must be in a folder “<exe directory>/data/ServerINI.txt”. This INI-File should be included to the exe and when building the data-folder should be created and should contain the file. If not it can be created afterwards when first starting the application.