

Compliant Kubernetes Service documentation has moved

Looking for Compliant Kubernetes Service documentation? All CKS docs have moved to their own site. [Read more here.](#)



[Datica Home](#) [Platform Dashboard](#) [Blog](#)

[Submit a request](#)

[Sign in](#)

[Datica Support & Documentation](#) > [Help](#) > [Mirth - Getting Started](#)

Compliant Kubernetes Service documentation has moved

Please note: You are not reading Kubernetes documentation. If you're looking for Compliant Kubernetes Service documentation, it has moved. [Read more here.](#)

Mirth Channel Overview

Rick Wattras - May 27, 2020 09:35

This chapter will break down all of the components of the Mirth channel configuration, with each subsection covering a tab of the Edit Channel console. It can be accessed by clicking the "New Channel" option in the pane to the left when in the Mirth Channels section.

Each channel in Mirth typically represents a uni-directional interface that performs a specific function following this basic formula:

- Receive messages
- (Optionally) transform the content or perform some logic on it
- Send messages

In this way, Mirth is usually the middleman between a sending application and a receiving application, and allows you to process data in a custom way while it's in transit. A common workflow, and one that Datica provides several channel examples for, is for a health system to send out bar-delimited HL7 messages via TCP that Mirth can then receive, translate to a more consumable format, and then send along the translated data to a proprietary API or custom application backend. Although the above formula is the root of almost all Mirth channel config, there is a lot of inherent ability to customize the handling and processing of data and that makes it a very useful tool.

Below we'll cover most of the options available in the Channel configuration that allow you to take advantage of the full scope of Mirth functionality.

Summary

The first section of the module that you'll be presented with upon clicking New Channel will be the "Summary" tab. Here you can configure some high-level options such as the channel name and the pruning settings. We'll detail some of the important settings available, but feel free to search out the Mirthconnect User Guide if you'd like a more in-depth look at the rest of the settings:

Edit Channel -

Summary | Source | Destinations | Scripts

Channel Properties

Name: ☒ Enabled ID: 31f0225d-9461-4119-ad2f-5f08342401d9

Data Types: ☒ Clear global channel map on deploy Revision: 0

Dependencies: Last Modified: 2017-10-10 10:56:43

Initial State: ☐ Store Attachments

Attachment: ☐ Store Attachments

Message Storage

Development

Content: All

Metadata: All

Durable Message Delivery: ☒ On

Performance:

☐ Encrypt message content

☐ Remove content on completion ☐ Filtered only

☐ Remove attachments on completion

Message Pruning

Metadata:

☒ Store indefinitely

☐ Prune metadata older than days

Content:

☒ Prune when message metadata is removed

☐ Prune content older than days

☒ Allow message archiving

(incomplete, errored, and queued messages will not be pruned)

Channel Tags

Tag
<input type="button" value="Add"/>
<input type="button" value="Delete"/>

Custom Metadata

Column Name	Type	Variable Mapping
SOURCE	STRING	mirth_source
TYPE	STRING	mirth_type

Channel Description

- Name = The unique channel name. Make sure it's specific so that in the case that you have similar channels there is no confusion. Be aware that there is a 40-character limit, and you can't use most non-alphanumeric characters like slashes, parentheses, and commas. Dashes and underscores are OK.
- Data Types = Clicking this button pops up the "Set Data Types" console where you can optionally modify the built-in data format presets that will pre-validate messages as the flow through the channel. For example, by default all new channels will assume that the data coming into the source listener, as well as the data being sent to the destination, is in HL7v2.x format and will perform some of its own validation at each point in the message traversal. However you can customize the data type at each step of message traversal, and you can even just set it to "Raw" in order to not have any validation be performed.

Set Data Types

☒ Single Edit ☐ Bulk Edit [Expand All](#) [Collapse All](#)

Connector	Inbound	Outbound
Source Connector	HL7 v2.x	HL7 v2.x
Destination 1	HL7 v2.x	HL7 v2.x

Inbound Properties **HL7 v2.x**

Serialization

☒ Parse Field Repetitions

☒ Parse Subcomponents

☐ Use Strict Parser

☐ Validate in Strict Parser

☒ Strip Namespaces

☐ Segment Delimiter \r

☒ Convert Line Breaks

Batch

Split Batch By: MSH Segment

Response Generation

Outbound Properties **HL7 v2.x**

Deserialization

☐ Use Strict Parser

☐ Validate in Strict Parser

☐ Segment Delimiter \r

Template Serialization

☒ Parse Field Repetitions

☒ Parse Subcomponents

☐ Use Strict Parser

☐ Validate in Strict Parser

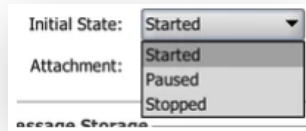
☒ Strip Namespaces

☐ Segment Delimiter \r

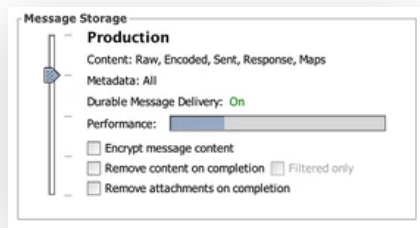
☒ Convert Line Breaks

- You can alter the data type by clicking the relevant down arrow under the Inbound and Outbound columns in the top pane. Other options include Delimited Text, DICOM, JSON, XML, EDI/X12, NCPDP, HL7v3.x, and Raw.
- Each step listed in the top pane can be selected and the relevant options will be presented in the bottom pane. For HL7 there's not usually much of a need to alter the default settings, but some helpful options for the other data types include:
 - Delimited Text: You can specify the column delimiter if it's something other than ',', and you can also set it to ignore header rows in each file if necessary
 - XML: By default Mirth is set to strip namespaces out of XML messages, but you can disable this here if the missing namespaces are causing issues during processing later

- Initial State = This drop down allows you to set the behavior of the channel when it is deployed. By default it will be Started (which basically means that it is up and actively running), but you could have it deploy but then Pause or Stop. You would then Start the channel from the Dashboard when ready.



- Message Storage = The slider on the left will set how much message content is retained in the database and available for view in the dashboard for each message. You'll see that as the slider goes down, less and less information is retained (for example, dropping it down to "Raw" retains only the raw message content and not any mapped variables or metadata) but the Performance of the channel is increased (the blue "Performance" meter indicates this). This means that it will potentially process messages faster and consume less memory, so if you don't care about retaining extra information but the speed at which the message is processed is important then adjusting this setting could prove useful.



- The "Remove content on completion" and "Remove attachments on completion" are useful in cases where you're dealing with large messages or images but don't want to take up a large amount of space in the database. Checking these will not store that data after it's successfully processed, and save space in doing so, but be aware that the removed content will not be viewable in the Mirth channel message dashboard.
- Message Pruning = This pane is where you configure the channel-specific pruning settings that dictate how long messages for that channel are saved in the database. You can set different timelines for Metadata (non-content related message info, like receive/response times) and Content. Note that if Pruning has not been enabled in the Mirth Settings → Pruning tab, then the settings will have no effect. Also note that pruning will not remove errored or queued messages - just successfully processed or filtered ones.



- If storage is a concern and you have a finite idea for how far back you will typically need to look for messages in the channel dashboard, then it's useful to configure pruning to limit the amount of disk the messages in the channel take up

Source

The Source tab is where you configure the settings that dictate how messages will be received by the channel and any transformation or filtering done during the initial processing.

Source connector

The main pane of the Source tab sets the functionality for how the channel will receive messages. There are several available "Connector Types" with each having their own settings and we'll cover the most common ones below. Before that, we'll quickly cover the settings that are universal to all connector types:

Source Settings

Source Queue:

Queue Buffer Size:

Response:

Process Batch: ☐ Yes ☒ No

Batch Response: ☐ First ☒ Last

Max Processing Threads:

- "Source Queue" = By default this is OFF, meaning that it will not queue messages up and will respond to each individually depending on the "Response" setting below. Switching it to ON will allow messages to queue up, which can be useful if the sender sends at a faster rate than the messages can be processed and a dynamic response generated. So for things like an HL7 message backlog, where the sender is just trying to send messages as fast as they can, it can be useful to queue them up on your end instead of having the sender wait to receive responses until each message is processed.
- "Queue Buffer Size" = This setting is only available when queueing is enabled by turning the Source Queue to ON. It dictates how many messages will be held in memory while queued. The default setting is usually fine.
- "Response" = This drop down menu allows you to set the response that will be sent back to the sender for each message, usually used to confirm receipt (ex: for HL7, an ACK message), confirm successful processing, or return a dynamic response relevant to the data received (ex: a custom HTTP response with relevant success/failure text). The options available by default are mostly only relevant to HL7, as using them with any other of the available datatypes will not return anything. But for HL7, using any of the "Auto-generate" options will automatically generate an ACK message as a response; the different settings just dictate *when* the ACK will be sent in terms of what part of the workflow (before or after processing). Not only can the ACK be used to confirm receipt but it can also be used to report a failure in processing by sending a "NACK". These are basically ACKs that include an ERR segment explaining why it was received but can also indicate if information was missing or incorrect. The non-automatic options can also be set to each Destination (we'll cover those shortly). Setting the response to a destination allows Mirth to automatically generate an ACK/NACK depending on the behavior of the Destination - if the sender would want to indication that the message was processed all the way to some other downstream destination, then it might be useful to have the response be dependent on that.
 - For HL7, it's standard to expect an ACK back as confirmation that a message has been received so remember to make sure you're set to respond to any HL7 senders!
 - There's also the concept of a "response map" variable, similar to a "channel map" variable except these will show up in the "Response" menu as additional options. You can therefore use them to set custom responses. The general format to create one is to call `responseMap.put("variable name", data);` in a Javascript writer.
- "Process Batch" = For the most part this is just relevant to reading in CSV files, where you can choose to parse each line in the file as its own message (set to "Yes") or parse the file in its entirety (set to "No" [default])
- "Max Processing Threads" = By default this is set to 1, meaning that one message will be processed synchronously at a time. Setting this to anything above 1 will basically allow you to 'multithread' the source handler and process multiple messages at the same time. Head's up that this increases memory consumption, and also this does not guarantee processing message order - the latter point is relevant for HL7, where you'd probably want to process messages in the correct order.

On to the different Connector Types and some of their useful settings:

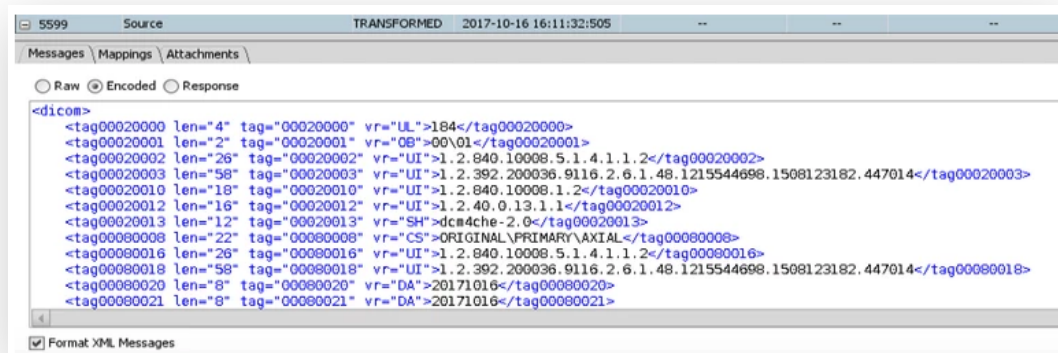
Listeners

These connector types simply receive incoming traffic via their individual protocols, and typically require you to open a specific port for the interface to listen on.

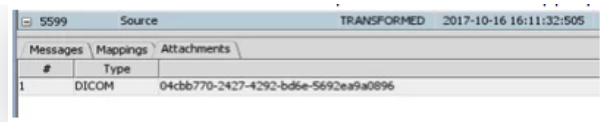
- Channel Reader = The default and most basic connector type, this option basically allows the channel to listen for messages sent from other channels within the same Mirth instance. In this way, it cannot be used as an externally-facing listener (unless chained to a real listener upstream) but is useful for test channels or internal-facing functionality like error handling.
- TCP Listener = Probably the most popular connector is the "TCP Listener", which listens for incoming messages sent over TCP - which is the industry standard for sending HL7 messages. The *Local Port* setting is probably the only thing you'll need to modify on the Source page, with the rest being left at their Default values. For Datica integrations, all TCP traffic sent/received to/from an external organization is transmitted across a site-to-site VPN connection between Datica and the other system. Because of that, you'll need to specify what ports you will be listening on or sending to when requesting a VPN connection from Datica. For more information about this, please contact Datica support. The only other TCP Listener setting that can be useful to understand is the *Transmission Mode*, which by default is set to MLLP but which can also be changed to "Basic TCP". MLLP is basically a message format protocol that acts as a wrapper for each HL7

message. It looks for a "start" byte" and an "end byte" at the beginning and end of each message in order to correctly parse the HL7. These bytes don't contribute to the actual content of the message itself. If the sender is not sending their HL7 messages wrapped in MLLP, then you may have to change this setting to accept Basic TCP, otherwise Mirth will reject the messages as not being in the expected format. Note that rejected messages will not appear in the message list of the channel dashboard, but the errors will appear in the logs.

- Note that for any port-based listener, if a port is already in use - either by a system process or by a separate Mirth channel - then the channel will not Start. It may Deploy, but will be in a "Stopped" state and an error will be logged indicating that the channel failed to start due to the port already being in use.
- HTTP Listener = Another popular connector is the HTTP Listener, which also primarily requires you to configure just the listening port. Because the Mirth service is running inside a Datica environment, the Mirth listeners aren't accessible to the outside world. In order to route external HTTP requests down to your Mirth instance, you'll have to configure your Service Proxy service to contain an nginx that forwards traffic sent to an external-facing URL down to your Mirth container. For more information on how to do this, contact Datica support. Some other useful HTTP Listener settings that you may want to configure are:
 - Binary MIME Types = This setting is pre-populated with a regular expression that if matched by the Content-Type of the HTTP request will base64-encode the message content, which renders it impossible to view in the dashboard without decoding it outside of Mirth. If that's something you'd like to have in place to limit the ability to directly view message content (and thus PHI in most cases), then feel free to leave this as-is. However if you'd like to be able to view message content, simply empty out the content of the text box so that it's blank and all Content-Type's are accepted without encoding the content.
 - Response Content Type = if you're sending back a custom response, you can specify the Content-Type here. By default it's "text/plain", but other examples would be "application/json" or "application/xml", etc
 - Response Status Code = The Mirth HTTP Listener is limited in its default HTTP Status Code response behavior - if the message fails to be received, or if the response is pulled from an ERROR status, then it will return a 500; otherwise it will send back a 200. If you'd like more control over the exact status code sent back, you can dynamically set a *channelMap* variable later in the message processing workflow and have it propagate to this field by using the Mirth 'environment variable' notation of *\$(variableName)*, where "variableName" is the name of the channelMap'd value.
 - One note on the "HTTP Authentication" settings: for Datica-based Mirth integrations, authentication is handled at the nginx level via the Service Proxy service, so you shouldn't have to mess around with this configuration.
- DICOM Listener = This Connector is primarily used for receiving messages via the DICOM protocol, which handles images transmitted from PACS or CT devices. Messages will come across as base64-encoded images, with XML-formatted metadata and usually an image attachment. Typically you should only need to modify the *Local Port* and *Application Entity* settings on the connector Source page. The port just determines where Mirth will be listening for the messages (similar to the TCP Listener), and the "Application Entity" is the same as the "AE Title" if you're familiar with PACS device functionality. It's basically just a alphanumeric identifier that must match to what the sender has configured as their AE Title, so make sure to coordinate with them while preparing to receive DICOM traffic. The rest of the settings can usually be left to the default, but feel free to peruse the Mirthconnect user guide if you'd like to further customize the listener.



- Example of XML-encoded DICOM metadata as seen in the Channel Message dashboard, when the "Encoded" radio button is selected. The various DICOM tags correspond to different data elements such as where the message came from, as well as information about the patient and procedure, but confirm these with the sender in order to parse the metadata appropriately.

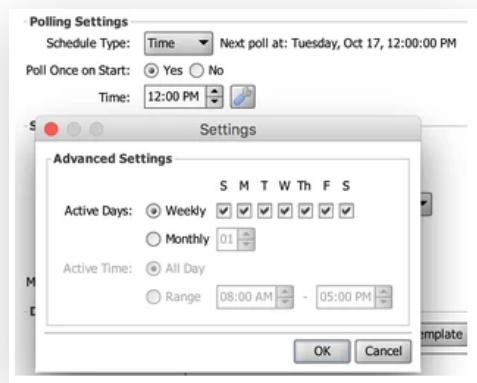


Source		
TRANSFORMED 2017-10-16 16:11:32:505		
Messages Mappings Attachments		
#	Type	
1	DICOM	04cbb770-2427-4292-bd6e-5692ea9a0896

- The "Attachments" tab is where you can sometimes view the image itself. Double-click the item in the list and a DICOM Image Viewer popup will appear and display the image. Note that some DICOM devices encode their images differently, so that sometimes you may see more than one 'Attachment' per message. If this is the case, the current version of Mirth can unfortunately not open these, but these messages can still be received and processed without issue.

Readers

These connector types are set to poll for data, or run a specific function, on a set interval rather than listen for real-time messages. The "Polling Settings" sub-section at the top is where you can set it run on a configurable interval (milliseconds, seconds, minutes, hours) or at a certain time of day. You can also click the wrench button for the advanced configuration that allows you to limit it to running on certain days or during active time windows:



Note that the Time is local to the server, which is in UTC. Also, you can set the poller to run once on start (when it's deployed) or just have it set to run first when it reaches its next polling time.

On to the connector list:

- Database Reader = This connector is pretty straight-forward and allows you to connect to a database and run a SQL query against it. The "Driver" options set what database type you're connecting to (PostgreSQL, MySQL, etc), while the URL points to the location of the database itself. If you'd rather run your query via Javascript instead of SQL, you can select the "Use JavaScript" radio button. Note that if using the standard SQL query, the results will be returned in an XML-formatted message wrapped in a `<result>` object. When building your query, it's useful to utilize the "AS" keyword in your SELECT statement so as to specify an easier name for identifying the values you're looking for in each column. For example, a SELECT statement like "SELECT db.column1 AS var1, db.column2 AS var2" will return a message similar to the following, which is straight-forward to parse:

•

```
<result>
  <var1>value1</var1>
  <var2>value2</var2>
</result>
```

- File Reader = Another popular polling-based connector is the File Reader connector, which seems pretty simple but has a lot of complex functionality available to it. The heart of the File Reader settings is the "Method" and subsequent Directory and path fields where you configure the location from which Mirth will be reading the files. Available methods are "file" (locally available on the Mirth host server), "ftp", "sftp", "smb", and "webdav". When a method is selected the path field is updated to reflect the chosen protocol like so:

File Reader Settings

Method: **sftp** **Test Read**

Advanced Options: Password Authentication / Hostname Checking Ask

Directory: **sftp://** /

Filename Filter Pattern: ***** ☐ Regular Expression

Include All Subdirectories: ☐ Yes ☒ No

Ignore . files: ☒ Yes ☐ No

Anonymous: ☐ Yes ☒ No

Username: **anonymous**

Password: *********

Timeout (ms): **10000**

- In the above example, we're using the SFTP method. In the "sftp://" fields, you'll populate the first box with the URL of the SFTP that Mirth will connect to, and the second box will be the path (if necessary) to the directory in which the files are located. If you only care about certain files, you can filter which ones Mirth will pick up by using the "Filename Filter Pattern". For SFTP only, you can also click the wrench button to open up the Advanced options and change things like authentication method (public key auth is available, you'll just need a public key file on the Mirth server to point to) or enabling Host Key Checking. For locations requiring a username/password, you can enter those in the relevant fields.
- Some convenient additional configuration revolves around the handling of the file once the content has been read in by Mirth:

After Processing Action: **Move**

Move-to Directory: **sftp://sftp.url.com/processed**

Move-to File Name: **\${originalFilename}**

Error Reading Action: **None**

Error in Response Action: **Move**

Error Move-to Directory: **sftp://sftp.url.com/errored**

Error Move-to File Name: **\${originalFilename}**

Check File Age: ☒ Yes ☐ No

File Age (ms): **1000**

File Size (bytes): **0** ☐ Ignore Maximum

Sort Files By: **Date**

File Type: ☐ Binary ☒ Text

channelName
channelId
DATE
COUNT
UUID
SYSTEM
originalFilename

- The "After Processing Action" options allow you to Move, Delete, or do nothing ("None") after the file content has been processed. Move or Delete are typically what you'd want to do, as selecting "None" will keep the file around after processing, which can cause it to be read in second time upon the next poll. When "Move" is selected, you can enter the location of the files to be moved to once it's successfully been moved. The same concept is applicable to the Error-related options. Note that for non-local directories, this means that you're moving the file to another location on the hosted server and *not* on your local server. When Mirth reads files from an external location, it only pulls the content into memory and doesn't pull the actual file across. One thing to be aware of is that if you're moving the file, you must enter a file name in the "File Name" field(s), otherwise the file will not be moved correctly. To just use the original file name you can drag and drop that variable from the list on the right, or just populate the field with `${originalFilename}` as seen above.
- Having "Check File Age" enabled (it's on by default) will cause Mirth to look at the age of the file and make sure its creation time is older than the specific amount. This helps to make sure that Mirth doesn't pick up any temporary files or partial files that in the midst of being written to the directory.
- If you are picking up binary files (images, encoded docs, etc) instead of text-based ones, simply select the "Binary" radio button at the bottom and Mirth will read the binary content in as a base64-encoded message.
- It was mentioned earlier, but in the case of .csv or other delimiter-based files with multiples entries it may be useful to enable batch processing by turning on the "Process Batch" option under Source Settings. This will process each line in the file as a single message, rather than the entire contents of the file as a single message.
- Javascript Reader = If you want to run some custom functionality that's not available in any of the built-in readers, then the Javascript Reader will allow you to write and run Javascript code that will be run on the set polling interval. To return any sort of message content, simply use the `return <value>` statement at some point in your code and that will populate a message that will be viewable in the dashboard.

Source filters

Mirth contains the ability to "filter out" the processing of certain messages depending on logic implemented in either the Source Filter or the Destination Filters. This is useful if you have distinct restrictions on what messages you'd like to receive, especially if that can be determined by content contained within the message itself. Filtering at the source allows you to evaluate every message that comes in on the interface; we'll touch on Destination filtering in the next section. How you actually implement the Filter logic remains the same at both locations though, so we'll go over that here.

While editing a channel and viewing the Source tab, click the *Edit Filter* option in the panel on the left. This opens the Source Filter configuration screen, and at first there won't be much of anything to look at. In order to begin adding a filter, click the *Add New Rule* link on the left and you'll see the top pane populate with a "Rule Builder" filter step. This is the easy built-in way to configure simple filter logic that allows you to select your options and set your values and Mirth will produce the code and inject it into the message processor behind the scenes - so this is useful if you only have simple requirements, don't require complicated filter logic, or would rather use the GUI-based configuration.

You can add multiple Rules, and can modify the operator with which their logic will be evaluated ('AND' or 'OR', like a logic gate) by double-clicking the black arrow in the Operator column of the row you want to modify. Once you have a Rule or two created, you can select them each and use the bottom pane to actually set the logic to match on. The "Field" box is where you'll insert the path to the field you want to evaluate in the message, so in the case of an HL7 message you'll insert something like `msg['MSH']['MSH.9']['MSH.9.1'].toString()`. As a tip, you can put an example HL7 message into the "Message Templates" box to the right, then use the fields listed in the "Message Trees" tab to drag-and-drop field paths from the message to the value field. You then choose the "Condition" (ex. "Equals", "Contains", etc) to match on and enter the values into the bottom pane by clicking the "New" button and entering the value(s) into the newly-created rows. Here is an example Source Filter:

The screenshot shows the 'Edit Channel - Source Filter' window. On the left is a sidebar with 'Mirth Views' (Back to Channel), 'Filter Tasks' (Add New Rule, Delete Rule, Import Filter, Export Filter, Validate Script, Move Rule Up), and 'Other' (Notifications, View User API, View Client API, Help, About Mirth Connect). The main area has a table of rules:

#	Operator	Name	Type
0	▼	Accept message if "msg['MSH']['MSH.4']['MSH.4.1'].toString()" contains "Facility1"	Rule Builder
1	AND	Accept message if "msg['MSH']['MSH.9']['MSH.9.2'].toString()" equals "A01" or "A04" or "A08"	Rule Builder

Below the table, the configuration for the selected rule is shown. The 'Behavior' is 'Accept'. The 'Field' is `msg['MSH']['MSH.9']['MSH.9.2'].toString()`. The 'Condition' is 'Equals'. The 'Values' list contains 'A01', 'A04', and 'A08'. There are 'New' and 'Delete' buttons for the values.

In this example, two rules are created: one matches on the MSH-4.1 "Sending Facility" field and only lets through messages that have an MSH.4 value equal to "Facility1"; the second rule looks at the Message Event Type MSH-9.2 field and only allows messages for events equal to A01, A04, or A08. Note that the operator chosen is an "AND", so both rules will need to pass in order for the message to not be filtered. In this way you can create filter rules purely by using the GUI options and let Mirth do the job of building the backend processing code. If you're curious to see the code it produces, you can click the "Generated Script" tab and it will actually show you the code statements generated by your settings, which is useful if you're code savvy and want to make sure the statements match to your intent. One note on adding values to the list: if you're matching on a string, enter the value as "value" (surrounded by quotes) in order for the logic to work correctly. This is due to how the code is generated, so the first Rule earlier translates to:

```
if((msg['MSH']['MSH.4']['MSH.4.1'].toString().indexOf("Facility1") != -1)) {
    return true;
}
return false;
```

- The if statement wouldn't evaluate correctly if "Facility1" was not in quotes.

If you'd like more hands-on control of the filter, or if it requires more complex logic outside of what's possible within the "Rule Builder" GUI options, then you can change the rule Type to be "JavaScript". This allows you to write the logic code yourself in the box below. As a general guideline, you can write any Javascript/Java (Mirth actually implements a version of Rhino, which is Javascript written in Java, so be aware of its slight differences in syntax) here as long as it eventually uses both *return true* and *return false* statements to allow the filter to be applied appropriately. As shown in the generated code sample above, you'll want to *return true*; to be run whenever you *don't* want the filter to be applied, and *return false*; when you *do* want it to be applied. It's a little bit confusing in their use of true/false but as long as your code results in one of those being returned then you can write and apply your own custom filter rules! Similar to the "Rule Builder" rules, you can still have multiple Javascript Rules that combine their logic via the "AND"/"OR" operators if you like. Feel free to play around with the "Rule Builder" Generated Scripts as a baseline to getting started with writing your own custom filters as that will show you the best ways to evaluate different conditions.

Messages filtered at the Source will not continue processing to the source transform or any of the destinations. They will show up as "FILTERED" in the message dashboard, instead of "TRANSFORMED" as for unfiltered messages. Note that filtered messages will still be pruned similar to normally processed message. Also, if you're accepting HL7 messages and have your Response settings set to anything other than "Auto-generate (Before processing)" then note that if the message is filtered it will send a NACK back as the response with a message saying the the message was rejected. If you don't want this to be the case, you may want to change it to "Before processing" or to apply filters later during the Destination processing (see below).

Source transform

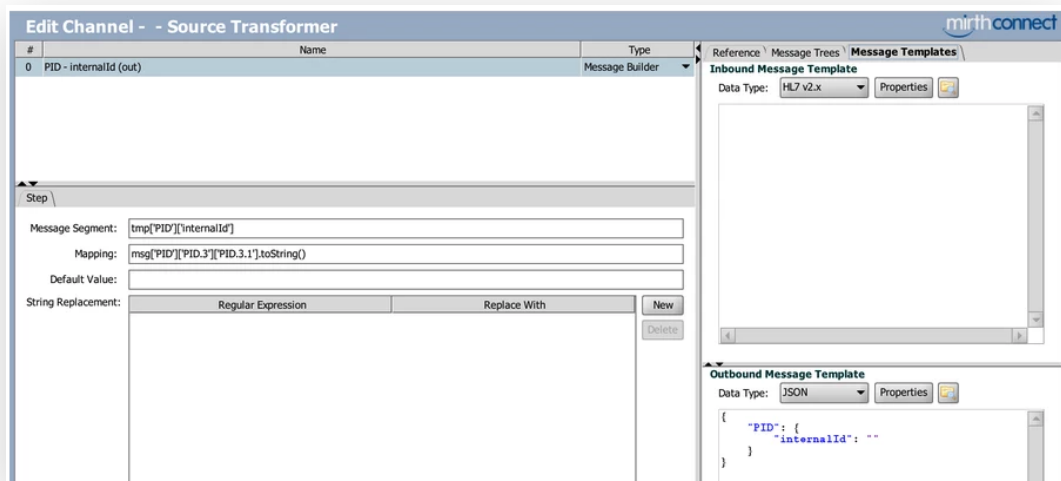
The Source transform is where you can build the translation code for how you'd like to map message data from one format to another. For Datica, this is where the majority of our HL7-to-JSON transformations are built; check out the Mirth OCI Github repository [here](#) if you'd like to import any of our pre-built transforms. Check out [Utilizing Datica Open Source Resources](#) if you'd like more information on that. If you'd like to build your own, use the information below to understand how to do so.

Similar to how the Filter described above can implement multiple "Rules", Transformers can create multiple "Steps", each containing a different Type of transform:

- Mapper = The Mapper type allows you to map a field value to various "Map" variables such as *Channel Map* (which can be pulled into and used in other locations within the current channel configuration), *Global Map* (which can be used in any channel within the Mirth instance), or *Response Map* (which can be used to populate a custom response for the current interface). The configuration options are pretty straightforward and work similarly to the filter's "Rule Builder". Here is an example Mapper step:

- This maps the MSH-4.1 value to a Channel Map variable called "sendingFacility". Because it is a Channel Map variable, it can be called from other locations within the current channel by calling "\${sendingFacility}" from within a JavaScript transformer or "\${sendingFacility}" from within a text box-based setting (ex. "Template" in the Destination)
- The free hand JavaScript code for setting each of the different Mappers yourself is:
 - Channel Map = channelMap.put("variableName", value);
 - Global Map = globalMap.put("variableName", value);
 - Response Map = responseMap.put("variableName", value);
- Much like the filters, you'll also see a "Generated Script" tab that you can use if you like, although Mirth will add some validation steps to the code that you may or may not need to include.
- Message Builder = If you have a defined outbound message structure that's easily mappable (ex. JSON, XML, or even simple HL7 messages) then the Message Builder may be of use to you. It allows you to directly map incoming message

data to an outbound message template all within the Mirth GUI and no actual code needed (Mirth generates it on its own, similar to the rest of the fields above). Once you select the "Message Builder" option as the transformer step type, the bottom pane will populate with three main fields: *Message Segment*, *Mapping*, and *Default Value*. The *Message Segment* field is where you set the outbound field path that you wish to map to, while the *Mapping* field contains the inbound field path. As mentioned before, you can populate the Inbound and Outbound message boxes under the Message Template tab and drag-and-drop values directly into the fields to auto-populate the full path. For Message builder steps, populating the Outbound Message Template allows you to have the most control in cases where there is a 1-to-1 mapping. Here's an example:



- In this example, we're mapping an inbound HL7 message field (PID-3.1) to an outbound JSON value (*PID.internalId*). Having the value already existing in the outbound message template allows it to automatically have a default value of "", as well as makes it easy to drag-and-drop the values into the text boxes to auto-populate the full path.
 - You can see that the outbound message is automatically stored within a Mirth-reserved variable called *tmp*, while the inbound is stored in a variable called *msg*. Knowing how to utilize both will be useful if you want to build a custom map, which we'll talk about below.
 - If you're building your outbound message by populating the Outbound Message Template (*tmp*), then know that if you want to get that message later during the Destination phase it will be stored in a Mirth-reserved Channel Map variable called *message.encodedData*. You'll see this as the default Template value for most Destination senders (in the form of "\${message.encodedData}", as it basically represents the message content as it is received at the time of hitting the Destination. If you would not have any transformation steps at all, the value would be the original message. Building a message with the Outbound Message Template however, will allow for the message that hits the destination to be the one that leaves the source transform - in this case, *tmp*.
- JavaScript = As described in the Filter section, the JavaScript step can be used to program custom JavaScript code that will perform mapping functionality (or whatever else you'd like it to do) when the built-in functionality will not suffice or would become cumbersome. Datica primarily relies on the JavaScript step type to build our standard mapping templates because it gives us the most flexibility in how we evaluate and populate our translated outbound messages. For more information about using the Datica templates, check out [Utilizing Datica Open Source Resources](#).
 - If you are interested in coding up your own transformer, here are some tips:
 - Utilize Channel Map variables to exchange values between steps
 - Try/Catch blocks are useful when making calls that may cause exceptions
 - Purposefully throw errors by using *throw("error")*
 - Manually log entries into the Mirth log by calling *logger.debug("content")*
 - Import code libraries with the *importPackage* syntax. Example:
"importPackage(Packages.org.apache.http.client);"
 - Note that Mirth Transformer steps have a memory limit for how large they can get. Check out this relevant FAQ here: [Is there a limit to how large a single Javascript transformer step can be?](#)
 - If you find yourself repeating the same code block a lot, maybe think about utilizing a Code Template. More info on those can be found in the subsection on code templates of [Utilizing Datica Open Source Resources](#)
 - Use the Inbound Message Template and corresponding Message Tree to drag-and-drop full paths rather than manually writing them out yourself

- Similarly, use the Outbound Message Template to build an outgoing message and use the Message Tree to validate that you're mapping the fields correctly

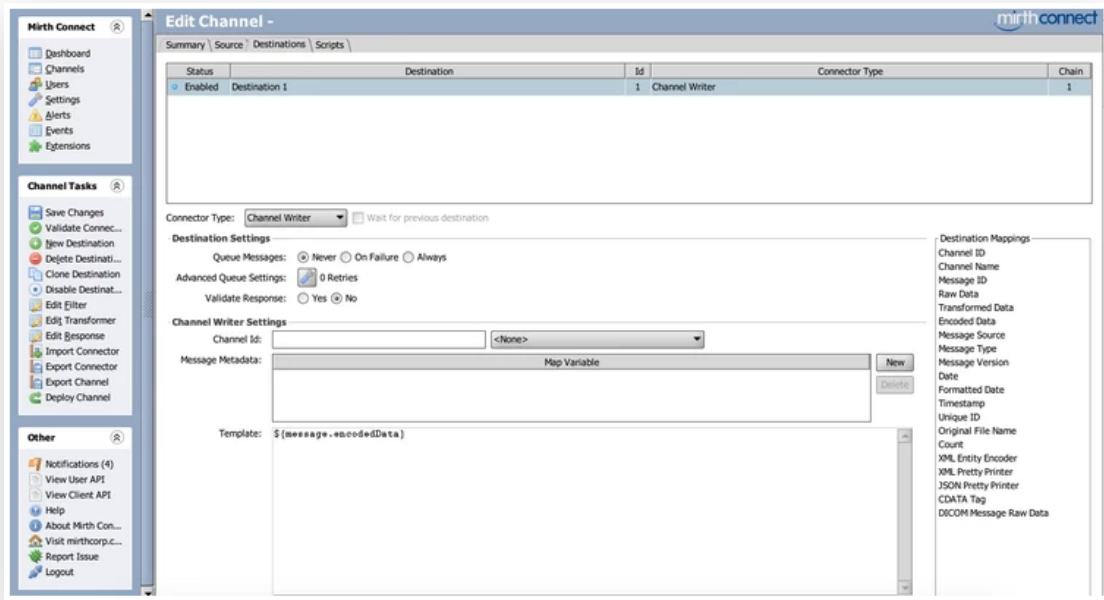
Much like channels themselves, individual Steps (or Rules for filters) can be exported/imported as well. So if you have a transformer that would be useful across multiple channels, but don't need to clone the channel in its entirety, you can simply export the step and import it into another channel via the Export Transformer/Import Transformer options when you have a step selected. You'll notice that in the Mirth OCI template channels, we utilized a new step for each segment of the HL7 message that the channel is meant to translate. Based on that model you can swap in and out transforms as necessary by either using the Steps within the channels themselves or by pulling from our repository containing all of our various transforms here: <https://github.com/daticahealth/Mirth-Transforms/tree/master/transforms>.

Destinations

The Destinations tab is where you configure the sending functionality for your Mirth channel: where it's going, over what protocol, and with what information. You can configure the channel to send to multiple destinations at once, or use the destinations like individual steps in a workflow. Much like the Source options however, each Destination can contain its own Filter, Transform, and Response (we'll get to that shortly) logic in addition to the actual configuration that sends the message.

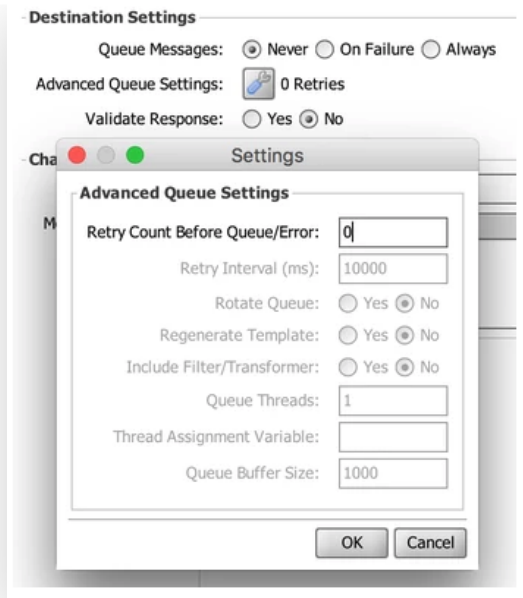
Destination connectors

When you first open the Destinations tab for a new channel, you'll notice that unlike previous configurations, there already exists a Destination listed in the pane at the top:



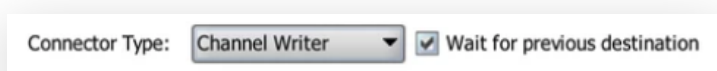
A Mirth channel must always have at least 1 active Destination, or else the message will have nowhere to go! By default, this Destination will not actually send to anywhere, as there is no channel specified in the "Channel Id" field under the *Channel Writer Settings*. This makes it useful for testing purposes if you want to play around with your transforms or filters without actually sending a message anywhere, so keep that in mind. That being said, there's a lot to unwrap here so let's jump right in.

In a way Destinations are similar to channels themselves in that you can have as many Destinations as you want, you can disable/enable them as needed, and as mentioned above they can each contain their own filtering and transformation logic as well. Similar to the Source options, they also can be of all different Connector Types, which we'll break down shortly. But also like the Source options, there are some settings that are shared by all of the Connector Types:



- **Queue Messages** = This setting sets the queueing behavior of messages as they enter the Destination for processing. A 'queued' Destination will basically continue to attempt to send the message until it goes through successfully. By default the "Never" setting will allow messages to flow normally, without any queueing behavior, and in cases where they fail to send they will simply be marked as ERROR in the dashboard and the channel will move onto the next message. The "On Failure" option will queue messages up if they fail to send and will continue to attempt sending until the message goes through successfully. "Always" will always queue the message before attempting to send. Both of the latter queueing-based options are useful if you want the message to continue attempting to send, so for instance if there is a temporary network issue with the receiver it will automatically send the message through when the issue is resolved. Also, they will still allow new messages to flow into the interface and attempt to send on their own destinations, so in this way they don't cause a backup of messages as the Source Queue does. Note however that if the issue that's causing the message to fail is due to something within the message content itself (so that it would be unchanged for each subsequent attempt) then you could possibly see the Destination end up in a QUEUED state indefinitely. Enabling the "Rotate Queue" option in the Advanced Queue Settings (more on that below) will allow for stuck messages to be sent to the back of the queue instead of holding up future messages, which should help to resolve the issue of a single message holding up the queue. However be aware that this does not guarantee that messages will be sent in the order you received them, if order is a concern.
- **Advanced Queue Settings** = Related to the above, you can use these settings to specify a number of automatic retry attempts and/or a retry interval at which to make those attempts. By default, the Destination will not retry at all as it is set to 0. When the "On Failure" or "Always" options are chosen above, you'll see this setting auto-populate with an interval setting of 10000ms indicating that it will retry every 10 seconds. You can modify these by clicking the wrench button to open the Setting popup (seen in the screenshot above). Other settings shown in that popup become available if you up the retry count from 0. Mostly the rest of these settings allow you to take advantage of multi-threading if you wanted to have multiple messages process through their destination at the same time. Similar to above where we described the "Rotate Queue" setting, be aware that enabling multithreading cannot guarantee that messages will be sent in the same order as they were received, so be careful with multi-threading if order is a concern.
 - If you find that a destination has occasional network blips that could cause messages to fail to send, it's definitely recommended to configure some automatic retries that will allow it to automatically wait and send again once the blip has abated. This will typically allow the message to go through immediately on the second try without any intervention from your support staff.
- **Validate Response** = This setting will use validation logic built into Mirth to determine if the destination returned a valid response or not. This is useful for confirming the receipt of HL7 ACKs returned from a receiving entity, but is not really used by the other data formats. If you want to parse other responses, check out the Destination Responses section below.

One final note on Destination settings available to all destinations: they will process in the order that they are listed. If you'd like them to process in parallel, you can uncheck the "Wait for previous destination" checkbox:



Now for the breakdown of each Connector Type and some of their more useful settings:

- Channel Writer = This is the default Connector Type, and much like the Channel Reader source connector it is only useful for communicating to other channels within the same Mirth instance. As mentioned above, this makes it useful for testing, but it can also be used in workflows where it may make sense to abstract out some sort of processing functionality by sending data to a different channel. We cover some examples of this in the "Chaining together channels" section of the Advanced Mirth Functionality chapter here: [Advanced Mirth Functionality](#)
 - The settings are pretty simple: you only need to specify a Channel to send to (or as mentioned before, you can leave it blank to send a message to "nowhere" for testing purposes) as well as the content of the messages, which you can populate as a template in the "Template" field. You can pull in Mapped variables from the *Destination Mappings* pane on the right, and add template content surrounding the variable content as well if you like.
- HTTP Sender = The most common Connector Type is the HTTP Sender, which provides a UI for building an HTTP request for sending your transformed data out to an external endpoint. The settings are pretty straightforward but allow for a variety of options:
 - URL = the HTTP or HTTPS address that you will be sending to. You only need to include the URI with the resource path, not any parameters if you're interacting with an API of sorts. Parameters can be added individually in a later setting. Note that if you supply an *https://* address then Mirth may warn you that "SSL Not Configured". This is because the HTTP Sender doesn't currently have the ability to pull in custom certs or exchange SSL certificates with the endpoint. If you need to use certificate-based auth, check out the "Certificate authentication" section in [Advanced Mirth Functionality](#). On the other hand, if you're using basic username/password or token-based authentication, this shouldn't be a problem and the warning can be ignored.
 - If you're sending through a VPN connection, you can use IP addresses here that are routable through your appliance. Something like *https://192.168.0.10:8123* can be routed on the appliance to a specific IP/port on the other side of the tunnel. Host files can also be used to direct an actual FQDN to the right IP on the appliance. For either option, just contact Datica support and ask them to help get that configured for you.
 - Method = Choose what HTTP method is needed for your request: POST, GET, PUT, and DELETE are your options
 - Send Timeout (ms) = Time in milliseconds for how long your request will wait for a response before erroring out. The message's Destination will be marked with an ERROR status in the dashboard (unless you enabled retries or queueing!)
 - Binary MIME Types = Similar to our recommendation for the Source setting of the same name, we find that just wiping out the content of this field is the easiest way to go. Otherwise, any response with a Content-Type within the default regular expression of *"application/.*(<?<|json|xml)|image/.*|video/.*|audio/.*" will be base64-encoded in the dashboard.*
 - Authentication = If you're authenticating yourself to the endpoint with a username and password, select Yes here and then fill in the Username/Password fields appropriately.
 - Query Parameters = Here you can create a list of Name/Value pairs that will be appended to the URL as parameters, as in some API or RESTful requests. Use the *New* button to the right to add a new pair
 - Headers = Add Headers to your request by creating Name/Value pairs for common headers like "Authorization: "<token>".
 - Content Type = This field is where you configure the Content-Type header for what content type (ex. "application/json") will be sent in the header of your request. The default value is "text/plain"
 - Data Type = Binary or Text, with Text being the default option. If you're sending binary data such as images or PDFs, choose the binary option.
 - Content = Finally, this text box is where you insert the content of the message that you'll be sending. Like most content fields in Mirth, you can populate this with a Channel Map variable or a built-in Destination Mapping, both listed in the pane on the right and drag-and-droppable into the Content field. The freehand value for a Channel Map is "\${variableName}"
 - Note that if you choose GET as your Method, you will not be able to send any Content in your request
- Example HTTP Sender filled out with information:

Connector Type: **HTTP Sender** ☐ Wait for previous destination

HTTP Sender Settings

URL:

Use Proxy Server: ☐ Yes ☒ No

Proxy Address:

Method: ☒ POST ☐ GET ☐ PUT ☐ DELETE

Multistep: ☐ Yes ☒ No

Send Timeout (ms):

Response Content: ☒ Plain Body ☐ XML Body

Parse Multipart: ☐ Yes ☒ No

Include Headers: ☐ Yes ☒ No

Binary MIME Types: ☒ Regular Expression

Authentication: ☒ Yes ☐ No

Authentication Type: ☒ Basic ☐ Digest ☐ Preemptive

Username:

Password:

Query Parameters:

Name	Value	
patientId	12345	<input type="button" value="New"/> <input type="button" value="Delete"/>

Headers:

Name	Value	
Authorization	WIND-K210-QPFD-0KEL	<input type="button" value="New"/> <input type="button" value="Delete"/>

Content Type:

Data Type: ☐ Binary ☒ Text

Charset Encoding:

Content:

Destination Mappings

- Channel ID
- Channel Name
- Message ID
- Raw Data
- Transformed Data
- Encoded Data
- Message Source
- Message Type
- Message Version
- Date
- Formatted Date
- Timestamp
- Unique ID
- Original File Name
- Count
- XML Entity Encoder
- XML Pretty Printer
- JSON Pretty Printer
- CDATA Tag
- DISCOM Message Raw Data
- H77JSON

- TCP Sender = The second most common connector type is the TCP Sender, which - you guessed it - sends messages over TCP. This is most useful within the Health IT sphere for sending outbound HL7 messages, but you can feasibly send any kind of message that you want as long as there is a listening port on the other end that can accept it! The main settings here are as follows:
 - Transmission Mode = If you're sending HL7, you'll most likely want to leave this as the default value of *MLLP*. This is an industry standard protocol message "wrapper" of sorts used to add start/end bytes to the HL7 message being sent out, and most receiving systems will expect them. If you're not sending HL7, you can feel free to change this setting to *Basic TCP*.
 - Remote Address = This is either the DNS name or IP of a receiving entity listening for TCP traffic. For most Datica integrations, this will likely be routed through a VPN appliance so you'll use some proxy IP like 192.168.0.10 that will be routed by Datica on the appliance to the actual destination IP on the other side of the tunnel. Just contact Datica support for help in getting that setup.
 - Remote Port = As the partner to the above setting, the port is where the receiving end is listening for your traffic. Similar to the IP routing, if your message is going through a VPN connection then Datica can route ports listening on the appliance through the tunnel to those listening at the receiver. Since the VPN appliance only has so many ports to listen on, and can't listen on the same port for more than one connection, it may be the case that sometimes Datica will provide you with a different port than the one listening on the receiver's side. Then on the appliance, that proxy port gets routed to the correct port on the other side of the tunnel. As mentioned above, if you're attempting to build an outbound TCP connection through a VPN tunnel, contact Datica and we will help you get that configured!
 - The "Test Connection" button next to the Remote Address can be used to see if Mirth is able to establish a TCP connection to the given IP/port. However, for connections through VPNs this only proves that it's able to connect to the appliance and not necessarily the receiver on the other end of the tunnel. If you'd like to confirm the full connectivity, Datica is happy to assist.
 - Keep Connection Open = This setting toggles the behavior once an outbound connection is established; by default the channel will only establish the connection while it's sending a message, then it will drop the connection. If you set this to Yes, Mirth will attempt to keep the connection open and established with the receiver for as long as you configure in the "Send Timeout" field below. Be aware however that if you're going through a Datica VPN appliance that there are network infrastructure pieces that may cut off the connection prematurely if it's been idle for too long, regardless of the setting here.
 - Note that setting "Send Timeout" to 0 will tell Mirth to attempt to keep the connection open indefinitely.
 - Response Timeout = Set how long (in milliseconds) the message will wait for a response before erroring out (or retrying/queuing if you have either enabled)
 - Data Type = Binary or Text, with Text being the default option. If you're sending binary data such as images or PDFs, choose the binary option.
 - Template = This text box is where you insert the content of the message that you'll be sending. Like most content fields in Mirth, you can populate this with a Channel Map variable or a built-in Destination Mapping, both listed in the pane on the right and drag-and-droppable into the Content field. The freehand value for a Channel Map is "\${variableName}". For outbound HL7, particular that which was built using an Outbound Message Template, you can use the default "\${message.encodedData}" value.
 - Here's an example of a TCP Sender with all of the options filled out:

The screenshot shows the 'TCP Sender Settings' configuration window in Mirth. The 'Connector Type' is set to 'TCP Sender'. The 'Transmission Mode' is 'MLLP'. The 'MLLP Sample Frame' is '<VT> <Message Data> <FS> <CR>'. The 'Remote Address' is '192.168.0.10' and the 'Remote Port' is '8123'. The 'Override Local Binding' is set to 'No' with a local address of '0.0.0.0' and port '0'. The 'Keep Connection Open' is set to 'No'. The 'Check Remote Host' is set to 'No'. The 'Send Timeout (ms)' is '0'. The 'Buffer Size (bytes)' is '65536'. The 'Response Timeout (ms)' is '15000' with an 'Ignore Response' checkbox. The 'Queue on Response Timeout' is set to 'Yes'. The 'Data Type' is 'Text'. The 'Encoding' is 'Default'. The 'Template' is '\$ {message.encodedData}'.

- **Database Writer** = The Database Writer connector is pretty straightforward and allows you to connect to a database and run either a SQL-based or JavaScript-based database query. These would be useful if you want to store inbound message content directly into a database. The options aren't very extensive and pretty much cover the database Driver (ex. MySQL, PostgreSQL, etc), the URL of the database, and some credentials to authenticate your query. If you need further control than that, you may consider custom-building your database connections and queries with the JavaScript writer, which we'll cover below.
- **File Writer** = If you want to write inbound message content to a file, either locally or on some external (s)FTP, then Mirth makes this pretty easy with the File Writer. Much like the Source Connector "File Reader", you just set where you'd like to write the file and what content to write and that's all there is to it. The Method options range from "file" (local directory on the Mirth server) to "sftp", which requires a username/password to access. The content of the file that you want to populate goes in the "Template" field, which again can be dragged-and-dropped from the pane on the right or custom-built as a template with dynamic content inside. Some other useful settings include:
 - **File Name** = Here you set what the filename will be for the file being written. In order to make it dynamic, use either a Channel Map variable or one of the Destination Mappings in the pane on the right. One example that includes the channel name, message ID, and a date/time could be something like `"${message.channelName}_${message.messageId}_${date.get('yyyy-M-d H.m.s')}.txt"`. That might translate to something like "Inbound Facility1 ADT_1223_2017-10-20 10.00.00.txt".
 - **Timeout** = How long (in milliseconds) to wait for a response from the directory/external system where you're writing the file
 - **File Exists** = Choose how you want Mirth to handle if a file with the same name that already exists in the location. "Append" will add the new content at the end of the file, while "Overwrite" will replace the old content with the new content. "Error" will log an error indicating that a file with the same name already exists. If you use a dynamic naming scheme utilizing the Destination Mappings as mentioned above, this shouldn't be a problem. However, we recommend choosing "Overwrite" as a standard option over the default "Append".
 - **File Type** = Binary or Text, with Text being the default option. If you're writing binary data such as images or PDFs, choose the binary option.
 - **Example File Writer** with some of the fields filled out:

File Writer Settings

Method: sftp Test Write

Advanced Options: Password Authentication / Hostname Checking Ask

Directory: sftp://examplesftputil.com / directory/path

File Name: %id%_\$(date.get('yyyy-M-d H.m.s')).txt

Anonymous: ☐ Yes ☒ No

Username: sftpuser

Password: *****

Timeout (ms): 10000

Secure Mode: ☒ Yes ☐ No

Passive Mode: ☒ Yes ☐ No

Validate Connection: ☒ Yes ☐ No

File Exists: ☐ Append ☒ Overwrite ☐ Error

Create Temp File: ☐ Yes ☒ No

File Type: ☐ Binary ☒ Text

Encoding: Default

Template: \${message.encodedData}

- Document Writer = An extension of the File Writer functionality, the Document Writer is useful if you want to generate formatted documents like PDFs or RTFs. However, the caveat is that you can only write these locally onto the Mirth server so you'll need some other process that can pick them up if you want to do something else with them once they're generated. In general, Mirth can write to any directories within the base install directory (*/opt/mirthconnect*) but if you need to create a subdirectory or would like to save files off to somewhere else, just contact Datica support and we can get that setup for you. The options here are limited but have a lot of power underneath:
 - Directory = The file path for where you want to save the generated document
 - File Name = The name of the file. Similar to with the File Writer, you can use either a Channel Map variable or one of the Destination Mappings in the pane on the right to keep it dynamic.
 - Document Type = PDF or RTF are your options
 - HTML Template = Here are the guts of the document being built, and where you can take advantage of the "template" aspect of the Writer. If you're familiar with HTML formatting, you can build an HTML framework directly in the field and inject content by inserting Channel Map or Destination Map variables. As long as it's valid HTML, Mirth will do all the work to use your HTML to generate the PDF or RTF document.
- JavaScript Writer = Finally, much like the JavaScript options for the Source filter and transform the JavaScript Writer exists for any other custom implementation that is either easier to manually write the code for or isn't possible with the existing built-in Connectors. Here you can use Mirth's Rhino (Java-based JavaScript) implementation to build your own sending functionality. Some examples of useful custom JS-based solutions might be API integration or HTTP requests that use certificate-based authentication. Here's a useful JavaScript Writer destination template for producing an HTTP request, adding options, and handling the response:

```
// Required Apache packages
importPackage(Packages.org.apache.http.client);
importPackage(Packages.org.apache.http.client.methods);
importPackage(Packages.org.apache.http.impl.client);
importPackage(Packages.org.apache.http.message);
importPackage(Packages.org.apache.http.client.entity);
importPackage(Packages.org.apache.http.entity);
importPackage(Packages.org.apache.http.util);
importPackage(Packages.org.apache.http.ssl);
importPackage(Packages.org.apache.http.conn);
importPackage(Packages.org.apache.http.conn.ssl);
importPackage(Packages.org.apache.http.client.config);

var httpClient = new DefaultHttpClient();

// Building the request
```

```

var queryURL = "https://someurlhere.com/path"; // Insert the intended web destination here
var httpPost = new HttpPost(queryURL); // Exchange out 'HttpPost' for other methods like 'H
httpPost.addHeader("Content-Type", "application/json"); // If sending JSON
httpPost.setEntity(new StringEntity($('hl7JSON'))); // Add the content to the request

// Adding a timeout to the request
var params = RequestConfig.custom().setConnectTimeout(60 * 1000).setSocketTimeout(65*1000).
httpPost.setConfig(params);

var statusCode, entity, responseString, resp;
try {
    // Execute the request and grab the response
    resp = httpClient.execute(httpPost);
    statusCode = resp.getStatusLine().getStatusCode();
    entity = resp.getEntity();
    responseString = EntityUtils.toString(entity, "UTF-8");

    // Save the response status code and content to Channel Map variables
    //      View them later in the message dashboard for review or troubleshooting
    channelMap.put("statusCode", statusCode);
    channelMap.put("responseString", responseString);

    // Throw an error if the status code is in the error range above the 200's
    if (statusCode >= 300) {
        throw("HTTP Response: " + statusCode);
    }
} catch(err) {
    logger.debug(err);
    throw(err);
} finally {
    resp.close();
}

```

There are a couple other Connector Types available so feel free to check those out on your own, or refer to the Mirthconnect user guide for more information.

Much like channels and rules/steps, Destinations can be imported/exported as well. When exporting a Destination, it includes all of the information configured in the Connector settings as well as any Filters, Transformers, or Responses built for that destination. We'll cover those latter functionalities next.

Destination Filters

These work almost exactly like the Source Filter described above, but they can be applied per Destination. This makes them useful if you want to send certain messages to one destination, but other messages to another - if there's a value that you can key off of in the message to dictate where it needs to go, simply add filters to both destinations to process each message through to the applicable destination. Messages filtered out at the Destination level will still show up as "TRANSFORMED" at the source in the message dashboard. If your response settings are set to response before processing or after the source transformer is done processing, then a successful ACK will be sent back (if dealing with HL7) even if the message is later filtered out.

Destination Transforms

Similar to the above, the Destination Transforms work pretty much like the Source Transform with the exception that you can have a different Transform for each Destination. This gives you the ability to send differently-translated data to separate destinations if needed, such as forking data to two locations in different formats.

Destination Responses

Unlike the various Source components, Destinations have the concept of "Responses". The "Response" is really just another Transformer, except that it is processed after the Destination is completed, and only if a response is returned to the Destination. In the case of HL7 transmitted via a TCP Sender, this could be used to parse the ACK for custom error parsing and alerting, among other uses.

It functions much the same as a regular Transformer, with multiple Steps and the same options when it comes to the Step 'Type'. For Destinations that do return a response, you should be able to utilize the Mirth built-in call `"response.getMessage()"` to obtain the content of the response that can then be parsed. Just make sure that the response you're parsing is formatted as expected, which can be set in the "Set Data Types" console linked from the Summary tab of the channel. Expand out the applicable Destination (click the plus sign) and you should see the Response element listed. Then make sure that the Inbound and Outbound fields of the Response element match with what you expect.

One gotcha to be aware of is that if you're HL7 ACKs back as responses then you should have the Response set to expect "HL7 v2.x" as the Inbound setting. Otherwise Mirth won't correctly validate the content of the response and will not automatically mark the message as "ERROR" if it fails to send or if it receives an NACK.

Scripts

The Scripts tab is the final tab at the top of the Channel editor screen. Here you can implement "scripts" that run at certain points in the channel workflow, as outlined below. They are all JavaScript writers, so you will have to custom code each of them if you want to modify the behavior at all. By default they all contain a `return` statement that is required so remember to keep that in place if you do end up modifying any of them!

Deploy

This script is for functionality that will run every time as soon as the Channel is Deployed. This could be useful if you want to load in a custom library, or set some baseline data in a Global Map each time the channel is deployed.

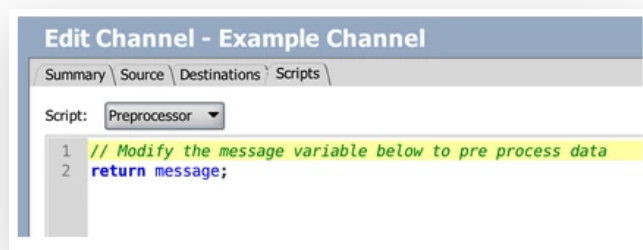
Undeploy

Basically the opposite of the Deploy script, the Undeploy script is run when a channel is undeployed. Some possible ideas for adding custom build here could be some sort of cleanup/garbage collection functionality or maybe to trigger some sort of external alert to notify somebody that the channel had been undeployed.

Preprocessor

For any functionality or logic that you would like to run before every single message - prior to it even hitting the Source filters and/or transformer - use the Preprocessor script. Some suggestions here could be sanitizing incoming messages for badly-formatted content, or setting a default response variable (ex. an HTTP status code) to an Error response, that is later modified to be a Success response if it traverses the channel workflow successfully.

CAUTION: it's mentioned above about leaving the `return;` code in place if you modify a script, but it's DEFINITELY required here. You'll notice that unlike the rest, the default statement here is actually `return message;`. Due to the nature of this script, in which you can modify message content prior to it hitting the Source transform, what it is doing is actually returning a message to the Source to be processed. The default `return message;` just passes the message along as it is received. If you modify the message at all, make sure that you're returning *something* at least or else the channel will act as if it didn't receive a message at all!



Postprocessor

Much like the Undeploy script is the opposite of the Deploy script, the Postprocessor script is pretty much the opposite of the Preprocessor script. If you want some functionality to kick off after every message, then use this script to do so. Similar to Undeploy, one useful application is cleanup/garbage collection if necessary. Another one, which piggybacks on a concept mentioned in the Preprocessor suggestions above, is to grab a Map variable that you may be altering throughout your channel

process and do something with it only after processing is complete. To go along with the HTTP status example, say that you initial set a variable to "500" in your Preprocessor and save it to a Global Channel Map. Later in a Destination, you successfully confirmed that your backend system processed the message, so you set the Map variable to "201". You can then have the Postprocessor script pick up the Map variable and set it in the "Response Status Code" field of the HTTP Listener Source settings.

Was this article helpful?



24 out of 25 found this helpful

