Example Channel

Creado y modificado por última vez por Nick Rupley el abr 27, 2017

Before continuing, read the FHIR Technology Preview 2 overview and user guide if you haven't already:

- Overview
- User Guide

This guide is separated into the following sections:

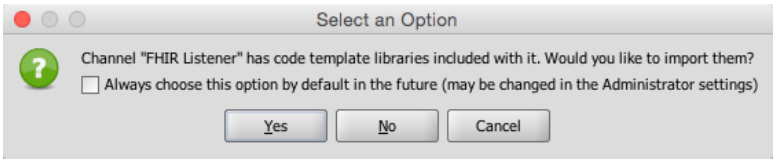
Importing the Example Channel

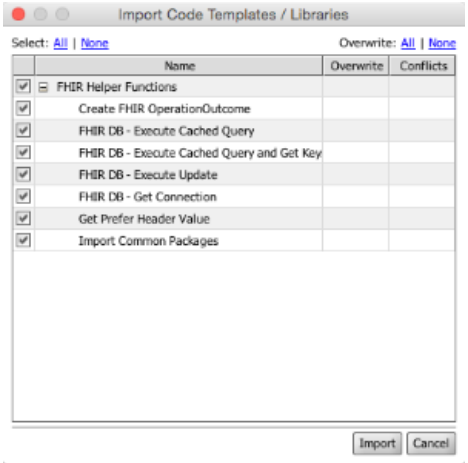
Download the channel here, according to the version of Mirth Connect you're using:

- Example - FHIR Listener (3.5.0).xml
- Example - FHIR Listener (3.4.2).xml
- Example - FHIR Listener (3.3.0).xml

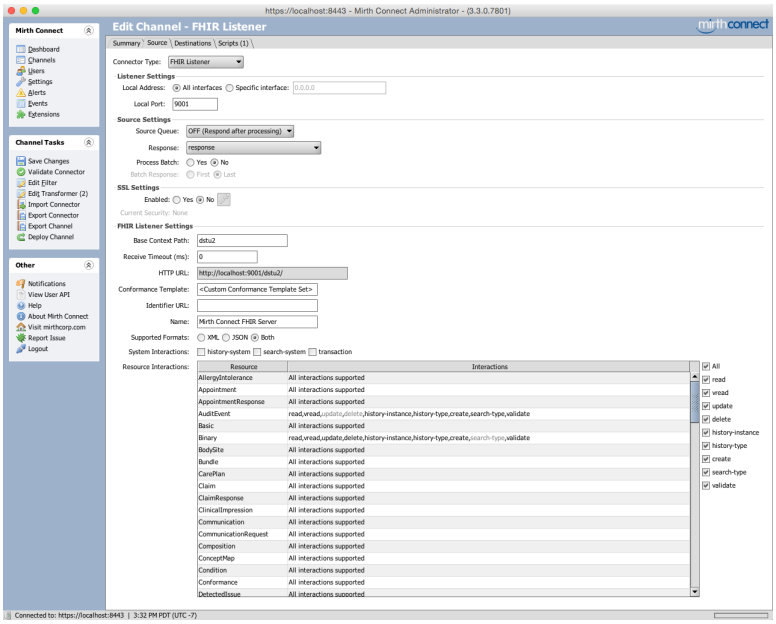
Then open the Mirth Connect Administrator, navigate to the Channels view, and click on Import Channel to the left. Select the channel XML file, and then click Open.

The channel comes bundled with a code template library, so make sure to choose Yes and then Import when it prompts you:





Then you'll be taken to the edit channel view:



Click on Save Changes to save the channel.

Creating the PostgreSQL Database

This example channel depends on a PostgreSQL database to store resource information. Once you create a schema (e.g. "fhirdb"), here are the other create statements you need:

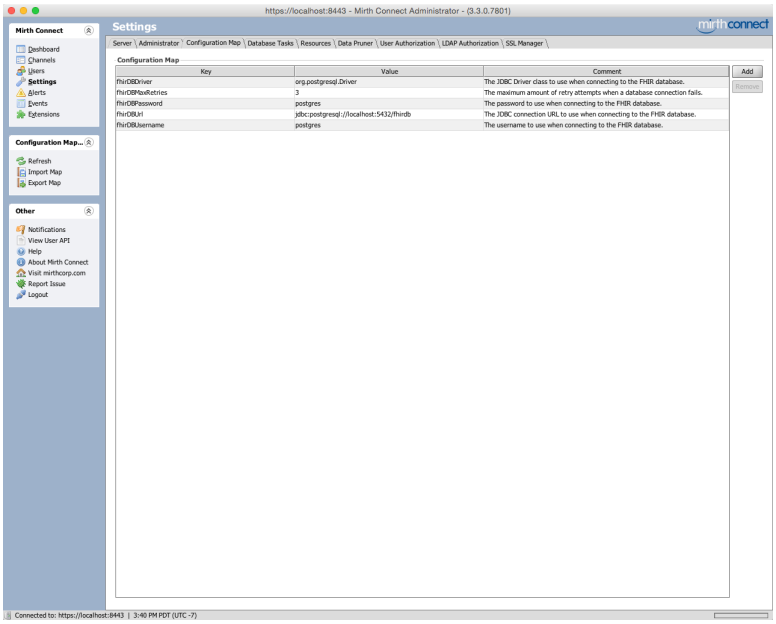
```
CREATE SEQUENCE resource_sequence
  INCREMENT 1
  START 1;

CREATE TABLE resource
(
  sequence_id bigint NOT NULL DEFAULT nextval('resource_sequence'::regclass),
  name character varying(255) NOT NULL,
  id character varying(255) NOT NULL,
  version integer NOT NULL,
  data xml,
  mimetype character varying(255),
  last_modified timestamp with time zone DEFAULT now(),
  deleted boolean,
  request_method character varying,
  request_url character varying,
  CONSTRAINT resource_pkey PRIMARY KEY (sequence_id),
  CONSTRAINT resource_unq UNIQUE (name, id, version)
);
```

Adding the Configuration Map Properties

The example channel also relies on some configuration map properties to store database connection information. If you don't already have configuration map properties set, you can just import this file:

```
# The JDBC Driver class to use when connecting to the FHIR database.
fhirDBDriver = org.postgresql.Driver
# The JDBC connection URL to use when connecting to the FHIR database.
fhirDBUrl = jdbc:postgresql://localhost:5432/fhirdb
# The username to use when connecting to the FHIR database.
fhirDBUsername = postgres
# The password to use when connecting to the FHIR database.
fhirDBPassword = postgres
# The maximum amount of retry attempts when a database connection fails.
fhirDBMaxRetries = 3
```

Otherwise, you can just add them to the configuration map table:



Make sure to change the connection information (URL, username, and password) as needed.

Notes on Implementation

First, note that certain interactions have been selectively enabled for certain resource types. For this example we'll only actually be implementing a few interactions, so this is largely for illustration. You can enable or disable interactions as you see fit, so that the generated conformance statement will reflect that support to clients.

On the source connector settings, we also have a custom "response" variable selected. This indicates that the FhirResponse object the FHIR Listener uses will be retrieved from the response map.

JavaScript Attachment Handler

Because the Binary resource type supports arbitrary binary payloads (like PDFs), we will want to create an attachment handler to extract those:

```
// If an incoming request includes a Binary resource, add an attachment for it
if (StringUtils.equalsIgnoreCase($('fhirType'), 'Binary') && StringUtils.isNotBlank(message)) {
    var contentType = $('headers').getHeader('Content-Type');
    var resource;

    if (StringUtils.equalsIgnoreCase(contentType, FhirUtil.getMIMETypeXML()) || StringUtils.equ
        resource = FhirUtil.isMIMETypeXML(contentType) ? FhirUtil.fromXML(message) : FhirUtil.f
        var attachmentId = addAttachment(resource.getContentAsBase64(), resource.getContentType
        resource.setContentAsBase64(attachmentId);
    } else {
        var attachmentId = addAttachment(message, contentType).getAttachmentId();
        resource = FhirUtil.createBinaryResource(attachmentId, contentType);
    }
    // Always return XML for Binary resources
    return FhirUtil.toXML(resource);
}

return message;
```

If the resource type is Binary and the interaction is create or update, the above code will add the binary data as an attachment, and return an XML serialized Binary resource with the attachment replacement token.

Source Transformer

Here we just have a couple of steps. First, we use destination set filtering to decide in advance which destination to send a message to. Each destination is named according to one of the possible FHIR interactions, like "create" or "update". Because the interaction of the request will be in the "fhirInteraction" source map variable, we can use that to directly filter on destinations:

```
var interaction = sourceMap.get('fhirInteraction');

if (interaction == 'operation' || interaction == 'validate') {
    // Operation destinations will have a name of "$name".
    destinationSet.removeAllExcept([sourceMap.get('fhirOperationName')]);
} else if (interaction.startsWith('history')) {
    // This will match history-system, history-type, and history-instance
    destinationSet.removeAllExcept(['history']);
} else if (interaction.startsWith('search')) {
    // This will match search-system and search-type
    destinationSet.removeAllExcept(['search']);
} else {
    // All other destinations should have a name equal to the interaction
    destinationSet.removeAllExcept([interaction]);
}
```

Then we map the Content-Type header to a channel map variable for later use.

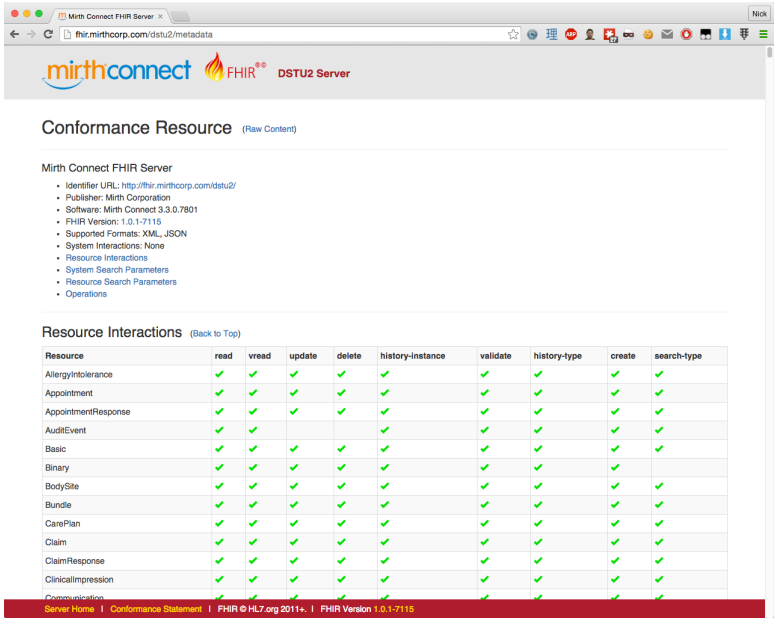
Destinations

We'll just look at one, the "create" destination. As the name implies, this destination will handle all create interactions that flow through the channel. It's a Database Writer that will take a resource posted to the channel and store it in a database (the one you created above). The JavaScript code in the destination simply inserts the resource into the database, and uses FhirResponseFactory to create a FhirResponse object. Then it places that response into the response map, with the key "response".

The other destinations in the channel are much the same. For example the "read" destination will use similar code to select resource data from the same database table, and return the data in an appropriate FhirResponse object. The "history" and "search" destinations are a little more complex because it involves selecting multiple resources and compiling them into a Bundle resource.

Sending Sample Requests

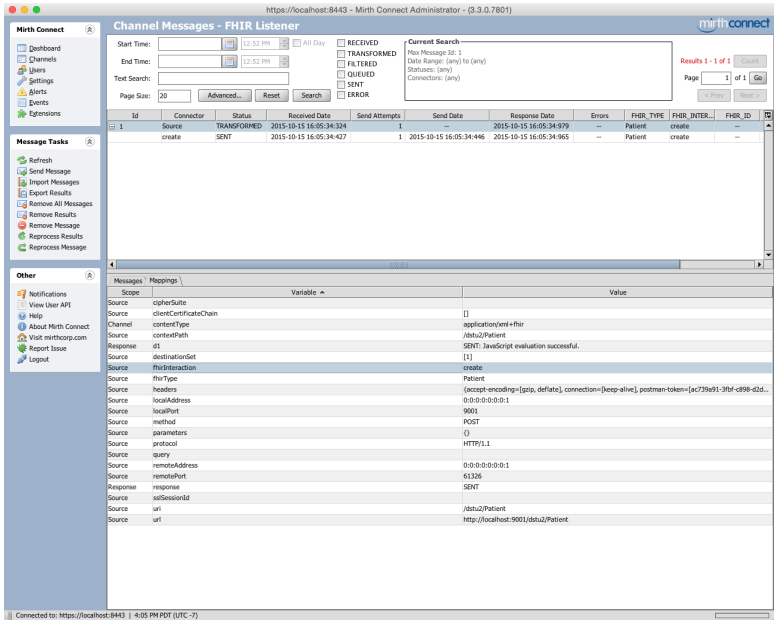
Once you've made any necessary tweaks to the channel or configuration map (like pointing it to your local database), save and deploy it. The FHIR Listener channel will be up and running, and you should be able to request the home page at the URL http://localhost:9001/dstu2, or the conformance statement at the URL http://localhost:9001/dstu2/metadata. Note that the IP, port, or base context path may be different depending on your source connector settings. If you request a resource (like the conformance statement) in a web browser, it will return the HTML template with the resource narrative (if available):



Creating a Patient Resource

After verifying the /metadata endpoint works correctly, try creating a new Patient resource. Doing so is simple, just POST a request to http://localhost:9001/dstu2/Patient with the Content-Type "application/xml+fhir", and the resource XML as the actual body. You can go here to get some example patient resources: Resource Patient - Examples.

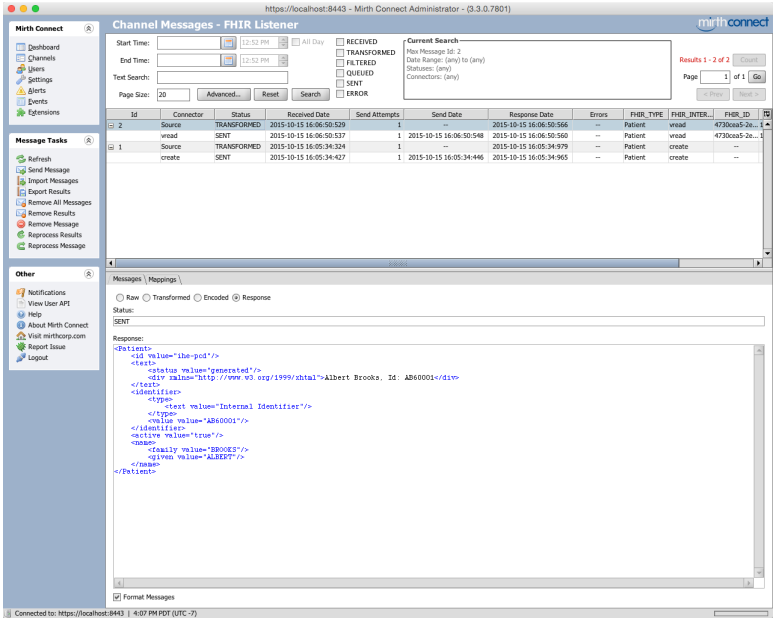
After sending the request, the channel should receive the message, and you can view it in the message browser:



Notice how the fhirType variable contains "Patient", and the fhirInteraction variable contains "create", which is correct. Back in whatever HTTP client you're using, you should have received a 201 (Created) status code, and also a Location header. The Location header contains a URL telling the client where to issue a "vread" interaction to retrieve the same resource you just created.

Reading a Patient Resource at a Specific Version

If you copy that URL and issue a new GET request to it, it should return the same resource XML that you POSTed earlier. Again in the message browser, you can view the vread request that came in, and verify the response data that was sent back to the client:



Binary Resources

You can also create and read Binary resources. Issue another POST request, but this time to the address http://localhost:9001/dstu2/Binary (again, URL may change depending on source connector settings). Use the Content-Type "application/pdf", and select a testing PDF for the actual HTTP payload. You should see the same "create" request in the message browser, but the PDF will be stored as an attachment instead, and the raw data will include a replacement token, like "${ATTACH:efe4cd42-de30-4e80-b1d4-1e15dbd646f9}".

Finally in the HTTP response, you should get the same Location header. If you copy that URL and issue a new GET request to it, it should return the same PDF that you created previously.

Next Steps

This sample channel is intended as a starting point to teach you the basics of FHIR resources/interactions and how the FHIR Listener works with them. From here, feel free to tweak the destinations or add more as you see fit. The draft standard basically just tells you what to do with resources (create them, update them, etc.), and you can use Mirth Connect to customize the actual implementation however you wish.

A 3 personas les gusta esto