HTML

HTML5 Server-Sent Events with Micronaut.io and Java

Building a simple, reliable messaging service

by Eric J. Bruno

April 2, 2020

I recently worked on an end-to-end IoT project where Micronaut was chosen as the microservice framework for the cloud-side implementation. Micronaut has built-in support for Kafka, RabbitMQ, and two HTML5 messaging paradigms: server-sent events (SSE) and WebSocket. Using them in a meaningful way, such as for publish/subscribe or queue-based messaging, takes a little effort. This article examines a simple, reliable messaging system built around Micronaut's SSE support.

As described on the project's website, Micronaut is a modern, JVM-based, full-stack framework for building modular, easily testable microservice and serverless applications. Based on dependency injection, aspect-oriented programming, and ahead-of-time compilation, Micronaut boasts very fast start times, good throughput, and low memory overhead. This makes Micronaut a good choice for cloud-based microservices where instances are spun up and down quickly.

For an introduction to Micronaut, check out Jonas Havers' article, "Building Microservices with Micronaut," and then peruse the well-indexed user guide.

For this article, download a full set of source code here. The file includes the three main projects:

- **MessageServer:** the Micronaut SSE server with `Queue` and `Topic Controller` classes, leveraging Micronaut's built-in SSE support
- **TemperatureSender:** a simulated temperature device that sends temperature readings over Micronaut SSE
- **Thermometer:** a web application with JavaScript to listen to the Micronaut server and receive temperature updates

The download file also includes my **SSELibrary** and two sample clients, **QueueSender** and **QueueReceiver**.

Now, let's dive into SSE messaging support.

Server-Sent Events Overview

HTML5 SSE is a server push technology that enables a browser (or any implementing application) to receive updates from a server over HTTP or

HTTPS. SSE works outside the browser as well, between applications written in any language. SSE doesn't require a separate server; it works over HTTP and HTTPS, it is firewall-friendly, and it's simple.

HTML5 SSE messaging is based on two main components: the text/event-stream MIME type, where text-based messages are sent according to a simple protocol, and the `EventSource` interface with event listeners to receive messages.

For more information on SSE, check out the W3C's HTML5 specification. You can also read my article, "HTML5 Server-Sent Events and Examples," for a sample implementation.

Programming SSE with Micronaut.io

To begin with SSE, create a Micronaut `Controller` class (essentially an HTTP listener) using the Micronaut `@Controller` attribute. In my project, I created two of them: one for queue-based messaging and one for topic-based (or publish/subscribe) messaging:

```
@Controller("/messageserver/api/")
public class QueueController extends Messenger {
    ...
}

@Controller("/messageserver/api")
public class TopicController extends Messenger {
    …
}
```

The Micronaut-based implementation sits between the sender and receiver applications, as shown in **Figure 1**.

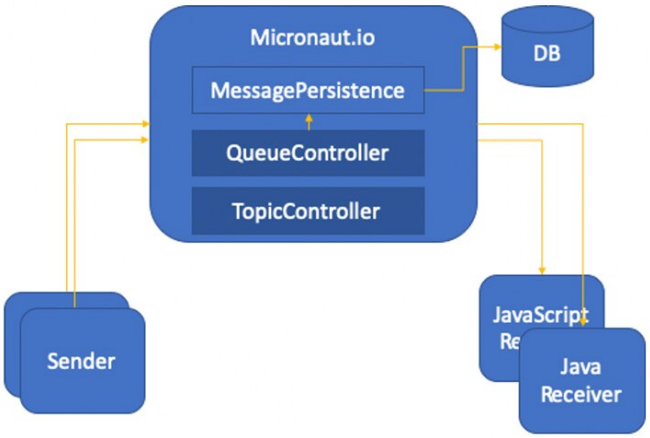

Figure 1. Micronaut's location between the sender and receiver apps

Each `Controller` specifies a URI path of `/messageserver/api`, which is combined with the base URI for your Micronaut server. I'll describe the `Messenger` base class a little later. For now, let's focus on the small amount of code it takes to send and receive events. **Listing 1** shows the code for receiving events and setting up an `@Get` REST endpoint that specifies the name of your endpoint—`Queue` or `Topic` in this case—along with a name for the queue to listen on.

Listing 1. The REST endpoint to receive queued SSE messages

```
@Get("/queue/{name}")
public Publisher<Event<String>> index(Optional<Stri
    // Determine queue to listen to
```

```
    Queue dest = getQueue( name.get() );

    return Flowable.generate(() -> 0, (i, emitter) -
        // Get the message first...
        Message msg = dest.getNextMessage();
        String data = new String( msg.getData() );

        // Then deliver it…
        emitter.onNext(
            Event.of(data)
        );

        // Finally delete it after delivery...
       dest.deleteMessage( msg.getId() );
    });
}
```

The `@Get` annotation indicates that this is an HTTP Get call handler, with `queue` and a queue name as part of the URL. When the endpoint is invoked, `getQueue` looks up the `Queue` destination object within a `HashMap` using the supplied name. If it's not there, it creates a new `Queue` object and inserts it into the `HashMap` using the supplied name.

Next, a Reactive Streams Publisher, which generates Event objects using a Flowable emitter, sends messages as they become available. With the queue paradigm, messages are saved until they are delivered to at most one receiver. As a result, after each message is delivered, the emitter deletes the queued message.

Sending messages to a destination from an application involves an HTTP Post (see **Listing 2**). The Micronaut Post handler (indicated with the `@Post` annotation) starts by looking up the destination by name.

Listing 2. The HTTP Post method that sends messages to a queue destination for delivery

```
// Content-Type: text/event-stream
@Consumes(MediaType.TEXT_EVENT_STREAM)
@Post("/queue/publish")
public HttpResponse queue( Session session,
                           HttpRequest<?> request,
                           @Body String data ) {
    try {
        HttpParameters params = request.getParamete
        String queueName = params.getFirst("name").
        Queue dest = getQueue(queueName);

        return processSend(dest, data);
    }
    catch ( Exception e ) {
        e.printStackTrace();
    }

    return HttpResponse.status(HttpStatus.UNAUTHORI
                               "Not authenticated")
}
```

The `@Consumes` annotation indicates that the `Post` expects a `text/event-stream` HTTP MIME type in the `Content-Type` HTTP header field. It's expected that the destination name is passed as an HTTP parameter. Once the destination object is retrieved, the message is routed to a live receiver or persisted if none exist.

Sending SSE Messages (Client Code)

To send HTML5 server-sent events in your application, you can use the `SSEDataPublisher` helper class (part of the `sselibrary` package you downloaded). This class works equally well with either queue- or topic-based messages and contains only one real method: `sendMessage`. It takes the URL of the message server that handles persisting and delivering messages, the message payload, and an authentication code for optional security.

According to the HTML5 SSE specification, the message payload needs to be formatted as a text string that contains three fields:

- The event type (such as heartbeat or message), for example, `event: message`
- A retry interval in milliseconds, for example, `retry: 30000`
- The data itself, for example, `data: "actual data here..."`

The text of each field must be terminated with a newline character, `\n`, and sent as part of the HTTP Post message, as shown in **Listing 3** (some code left out for brevity) with an extra newline added to the end of the data field.

Listing 3. The sendMessage method sends an HTTP Post according to the HTML5 SSE specification.

```
String event = "event: message\n";
String retry = "retry: 300000\n";
data = "data: " + data + "\n\n";

URL url = new URL( uri.toASCIIString() );
HttpURLConnection urlConn =
    (HttpURLConnection)url.openConnection();
urlConn.setFixedLengthStreamingMode(
    event.length() + retry.length() + data.length()
urlConn.setDoOutput(true);
urlConn.setDoInput(true);
urlConn.setRequestMethod("POST");
urlConn.addRequestProperty("Content-Type", "text/eve
urlConn.addRequestProperty("Authorization-Info", au

PrintWriter out = new PrintWriter( urlConn.getOutput
out.write(event);
out.write(retry);
out.write(data);
```

Of particular importance is `HttpURLConnection.setFixedLengthStreamingMode`, which enables the streaming of the HTTP request body without internal buffering. This must be set to the total message payload length, including the newline characters. Next, the HTTP `Content-Type` and optional `Authorization-Info` header fields are set. Finally, the data is written into the body of the Post. The message will be received and processed by the message server (described later).

Receiving SSE Messages (Client Code)

Receiving HTML5 server-sent events in your application is straightforward with the `SSEDataSubscriber` helper class as part of the `sselibrary` package. To use it, first implement the `SSECallback` interface, which defines a single method, `onMessage`, through which messages are delivered (see **Listing 4**). Next, create an instance of the `SSEDataSubscriber` class, providing the URL of the message server, whether it's a `Topic` or `Queue`, and an optional authentication string in the constructor.

Listing 4. Using the helper classes to receive SSE messages in your application

```
SSEDataSubscriber sse = new SSEDataSubscriber(
        serverURL, SSEDataSubscriber.DestinationType
sse.subscribe(destinationName, this);

// ...

@Override
public void onMessage(String queue, String data) {
    // ...
}
```

This is all it takes to implement a listener in your application. Let's take a deeper look at the `SSEDataSubscriber` helper class now.

Inside the SSEDataSubscriber class. The `SSEDataSubscriber` class abstracts and hides the HTTP mechanics to listen for SSE messages. Depending upon the destination type—`Queue` or `Topic`—the constructor (shown in **Listing 5**) appends the proper API path to the message server's REST endpoint URL.

Listing 5. The constructor forms the proper REST endpoint URL.

```
public SSEDataSubscriber( String serverURI,
                          DestinationType type,
                          String authCode ) {
    this.authCode = authCode;
    if ( type == DestinationType.QUEUE ) {
        this.serverURL = serverURI + "/api/queue/";
    }
    else {
        this.serverURL = serverURI + "/api/topic/";
    }
}
```

Next, when the client application calls `subscribe`, the supplied destination name and callback are stored, and the `Thread` is started. Execution moves to the `Thread.run` method, as shown in **Listing 6**.

Listing 6. The SSEDataSubscriber extended Thread.run method implementation

```
URL url = new URL(serverURL);
URLConnection conn = url.openConnection();
conn.setDoOutput(true);
conn.setConnectTimeout(0);

BufferedReader rd =
    new BufferedReader(
        new InputStreamReader( conn.getInputStream(

String line;
while ((line = rd.readLine()) != null) {
    if ( line != null && line.length() > 0 ) {
        // Did we get a heartbeat or useful data?
        if ( line.startsWith(":") ) {
            // heartbeat message...
        }
        else if ( line.startsWith("data:") ) {
            // Received data, send to the client's c
            if ( callback != null ) {
                callback.onMessage(destination, line
            }
        }
```

```
    }
}
```

A connection to the message server is created within this thread specifically to handle messages for the given destination. Once an HTTP message is received, it's determined to be either a heartbeat (indicated by an empty message) or one that contains actual data (indicated by the presence of the text `data:`). Data is delivered to the client asynchronously via the `onMessage` method on its supplied callback.

Now, let's go back and examine how the message server microservice (implemented with Micronaut.io) handles and delivers messages.

The Messenger Base Class

Returning our attention to the `QueueController` and `TopicController` classes, note that both inherit from the base class, `Messenger`. It's here that the `processSend` method (as referenced earlier in **Listing 2**) is implemented for both `Topic` and `Queue` destination types (see **Listing 7**). First, the data is split by newline characters (remember those were added when the SSE message was sent, per the specification).

Listing 7. The Messenger.processSend method within the message server microservice

```
public HttpResponse processSend(Destination dest, St
    String[] lines = data.split(System.getProperty('
    try {
        for ( String line: lines) {
            if ( line.contains("event:")) { }
            else if ( line.contains("id:") ) { }
            else if ( line.contains("data:") ) {
                int start = line.indexOf("data:")+"
                data = line.substring(start).trim()
                dest.addMessage(data);
            }
        }

        return HttpResponse.ok(dest.getName());
    }
    catch ( Exception e ) {
        return HttpResponse.serverError(e.toString(
    }
}
```

The message `data:` fields are pulled from the message text, and the destination's `addMessage` method is called with the message data. This method is defined in the `Destination` abstract base class but differs in implementation in both of its extending classes, `Queue` and `Topic`. Let's take a look at these now.

Inside the Topic class. The workings of a `Topic` destination are simple: Each message sent is delivered to every active listener, a one-to-many relationship (see **Figure 2**).

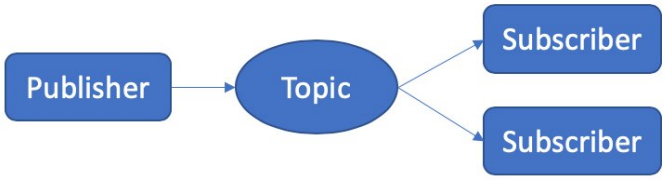

Figure 2. With topic-based publish/subscribe messaging, each message is delivered to every active subscriber.

When a message is sent, the `addMessage` method creates a `Message` object to encapsulate the message payload (text), stores the `Message` in the `Topic` object's `lastMessage` member variable, and signals all threads waiting on the `Topic`'s monitor object, as shown in **Listing 8**.

Listing 8. Handling an incoming topic message

```
public boolean addMessage(String msgData) {
    Long messageId = System.currentTimeMillis();
    Message msg = new Message(messageId, msgData);
    lastMessage = msg;

    // Notify ALL listeners of the message
    synchronized (topicMonitor ) {
        topicMonitor.notifyAll();
    }
    return true;
}
```

Every client that calls the message server's REST endpoint to receive topic messages ends up calling `getNextMessage` on the `Topic` class, where it waits on a monitor until it's signaled (see **Listing 9**) when a message is available.

Listing 9. The Topic.getNextMessage method waits until a message arrives for the destination.

```
public Message getNextMessage() throws InterruptedE
    synchronized ( topicMonitor ) {
        topicMonitor.wait();
    }
    return lastMessage;
}
```

That's it for topics; the `Queue` class, however, is more involved.

Inside the Queue class. The biggest differences between a `Topic` and `Queue` are that with a `Queue`

- Messages must be delivered to at most one listener (see **Figure 3**).
- Messages must be stored even when no clients are listening for eventual delivery.

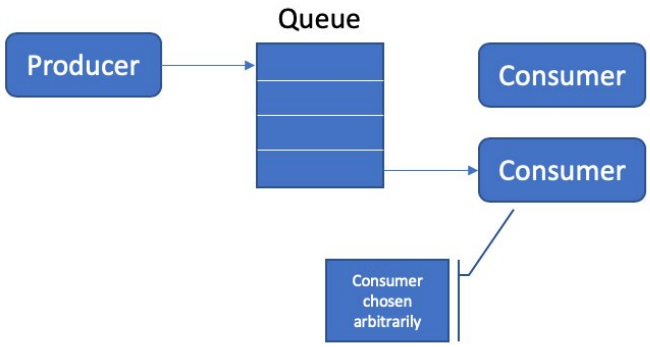

Figure 3. With a queue, each message sent is delivered to exactly one listener.

`Queue.getNextMessage` is invoked when a client calls the message server's REST endpoint (shown earlier in **Listing 1**). The messages themselves are never kept in memory; only message IDs are (see **Listing 10**). Because queued messages can stay in a queue indefinitely —until a listener finally shows interest in the associated queue—storing them all in memory could potentially consume all of it. The message data is persisted instead.

Listing 10. The caller is blocked while waiting for the next queued message ID on the destination.

```
public Message getNextMessage() throws InterruptedE
    // Blocking call
    Long messageId = messageIds.take();

    // Load the message data using the ID
    Message message = persistance.getMessage(getNam
    return message;
}
```

The `messageIds` object is implemented as a `java.util.concurrent.ArrayBlockingQueue`. The call to `take` is blocked until a queued entry is available, at which time the code removes and returns the head of the queue to only one blocked caller. With the message ID in hand, the message payload is retrieved from the persistent store (part of the reliability of queue-based messaging). Finally, once the message is delivered, its message ID is deleted along with the persisted message body.

Implementing Reliable Messaging

The interface `MessagePersistence` is defined to hide the details of how messages are actually persisted. The `Queue` object uses the Factory pattern to obtain an instance of a persistence implementation (see **Listing 11**).

Listing 11. The Queue object uses the Factory pattern to get the persistence implementation.

```
public class Queue extends Destination {
    final protected MessagePersistance persistance =
MessagePersistanceFactory
            .getInstance().getMessagePersistance();

        //...
}
```

The Factory pattern can be configured (via dependency injection, a configuration file, an environment variable, and so on) to load any persistence implementation, so long as it implements the `MessagePersistence` interface. Let's examine one as an example.

Using a NoSQL database for persistence. The `MessageNoSQL` class (in the download package) implements the `MessagePersistence` interface and uses Oracle NoSQL Database to store and retrieve messages by message ID. Because the `QueueController` class uses the Factory pattern and depends only on this interface, you can easily swap out implementations, such as a cloud-based NoSQL database.

Messages are stored using name-value pairs, where the *key* (the name) is the destination name and message ID combination. The *value* is the message payload, encoded as a byte array (see **Listing 12**).

Listing 12. Saving messages to the NoSQL datastore

```
public boolean saveMessage( String destinationName,
                            Long messageId,
                            String message) throws I
    String idStr = messageId.toString();
    store.put( Key.createKey(destinationName, idStr
               Value.createValue(message.getBytes()
    return true;
}
```

Retrieving a message is just as straightforward (see **Listing 13**). First, the key is assembled and used to retrieve the `Value` object. This object, if found, is used to retrieve the stored byte array representing the message payload. The result is transformed back into a `Message` object and returned.

Listing 13. Retrieving the message payload from the key (destination name and message ID)

```
public Message getMessage(String destinationName, Lo
    String idStr = messageId.toString();
    Key key = Key.createKey(destinationName, idStr)
    ValueVersion value = store.get(key);
    if ( value == null || value.getValue() == null
        return null;
    }

    Value val = value.getValue();
    String data = new String( val.getValue() );
    return new Message( messageId, data);
}
```

When the message server is first started, it uses the NoSQL database to load all stored message IDs back into memory. To do this, first `QueueController` iterates all of the destination's names associated with persisted messages (see **Listing 14**). The call to `getQueue` then creates `Queue` objects for each destination name (only queues are persisted, so this is safe).

Listing 14. Retrieving all destination names from the NoSQL database

```
private void loadSavedMessages() {
    ArrayList<String> queueNames = messageDB.getStor
    for ( String queueName: queueNames ) {
        // Get the queue (loads all queued messages
```

```
            Queue dest = getQueue(queueName);
        }
    }
```

The `getQueue` method (implemented in the base class, `Messenger`) constructs a `Queue` object with the given destination name, which loads all message IDs for that queue from the database in the constructor (see **Listing 15**).

Listing 15. Loading all message IDs for the queue from the database

```
public Queue(String name) {
    // load message IDs
    ArrayList<Long> ids = persistance.getMessageIds
    if ( ids != null ) {
        this.messageIds.addAll( ids );
    }
}
```

In this sample implementation, the database is assumed to be running locally (127.0.0.1) on port 5000, with the name `kvstore`. You can override this via a configuration file.

Running Oracle NoSQL Database. To run the SSE message server code for this article, download and install the Oracle NoSQL Database Community Edition. After installing the database, modify the `config.xml` file to set `hostname` for your computer and set `registryPort` to 5000. To run the database, I use the following command:

```
> java -jar lib/kvstore.jar kvlite -secure-config d
```

The last parameter disables security to make it easier to run in this example implementation, but it should not be used in a production environment. Once the database is started, you'll see output similar to the following:

```
Opened existing kvlite store with config:
-root ./kvroot -store kvstore -host Dolce -port 500
```

You can now move on to running the SSE message server, as shown in the next section.

An End-to-End Demonstration

You can start the SSE message server with Micronaut via the following command (remember to start the NoSQL database first):

```
> java -jar target/MessageServer-1.0-SNAPSHOT.jar
```

If the command is successful, you should see log output ending with a line similar to this:

```
12:53:28.857 [main] INFO io.micronaut.runtime.Micro
```

To implement a queue receiver, use the helper class `SSEDataSubscriber`, which takes the Micronaut server URI, the destination type (`Queue` or `Topic`), and an optional authentication code as parameters in its constructor. Once it is created, listen on a queue by calling the `subscribe` method (see **Listing 16**).

Listing 16. Subscribing to a queue

```
SSEDataSubscriber sse = new SSEDataSubscriber(
        serverUrl, SSEDataSubscriber.DestinationType
sse.subscribe( queueName, this );
```

In this case, the calling class implements the `SSECallback` interface and passes a reference to itself in the call to `subscribe`. As messages arrive on the queue, the object's `onMessage` method will be called with the payload. To see how this is achieved, review **Listing 6** earlier in this article.

To send messages to a queue, use the `SSEDataPublisher` class (covered earlier in **Listing 3**), as shown in **Listing 17**.

Listing 17. Sending messages to a queue

```
String url = serverURI + "/api/queue/publish?name="
SSEDataPublisher.sendMessage(url, data, authCode);
```

That's all it takes! Because the same helper classes work to send and receive messages to queues and topics, the Java code for both types of applications is very similar. Let's take a look at how to create a JavaScript listener to display dynamically updating data inside a web application.

Implementing a JavaScript SSE listener. This final sample uses a topic to send simulated temperature updates, updated dynamically on a web page using SSE. The Java temperature sender is similar to the queue sender explored in the previous section. The listener, however, is JavaScript code embedded within a simple HTML-based web page.

First, create the `EventSource` instance, as shown in **Listing 18**.

Listing 18. Creating an EventSource instance

```
var source = new EventSource(uri+"/messageserver/ap
```

Next, set up an error handler to help debug and reset the connection, as shown in **Listing 19**.

Listing 19. Setting up error handling

```
source.onerror = function(event) {
    console.log("SSE onerror " + event);

    // Wait 1 second and reconnect
    setTimeout(function() { setupEventSource(); }, 1
}
```

Finally, implement the message listener function (where the payload is delivered), as shown in **Listing 20**.

Listing 20. Implementing the message listener function

```
source.onmessage = function(event) {
    var tempGauge = document.getElementById('tempera
    tempGauge.innerHTML = event.data;
}
```

In this example, `tempGauge` is an HTML `div` element used to display the updating temperature value. When the SSE message server is running, with a sender publishing data to the temp1 topic, the web page will look similar to **Figure 4**.

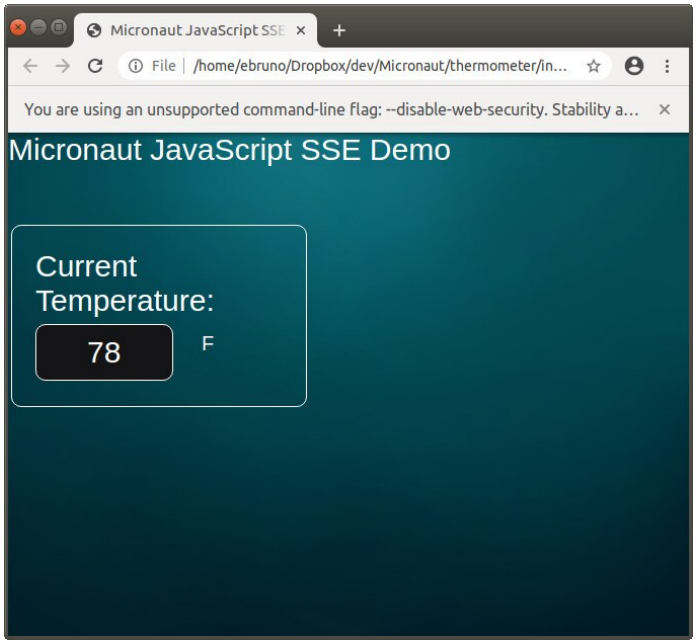

Figure 4. A web application with simple JavaScript to update the temperature reading

Note that due to cross-site scripting security, browsers such as Chrome will block data from URLs other than the one serving the web page. There are ways to deal with this, but the easy way for development purposes is to start Chrome from the command line with the following parameters:

```
google-chrome --disable-web-security --user-data-di
```

On Windows, replace `google-chrome` with `chrome.exe`.

As an even simpler example, you can open your browser normally and then enter the proper URL for a valid SSE destination, such as `http://localhost:8080/messageserver/api/topic/topic1`. As a result, the browser will append each data update to the page, although you'll eventually need to scroll to see the latest updates.

Conclusion

In this article, I demonstrated how to build a simple, reliable messaging system that is built around Micronaut's SSE support and supports queue-based messaging as well as topic-based (or publish/subscribe) messaging. Because Micronaut provides very fast start times, good throughput, and low memory overhead, it is a good choice for cloud-based microservices where instances are spun up and down quickly.

Eric J. Bruno

Eric J. Bruno is a lead real-time engineer at Perrone Robotics, where he's teaching cars to drive themselves. He has 25 years' experience in the information technology community as an enterprise architect, developer, and analyst with expertise in large-scale distributed software design.

Share this Page