Web Client Guide

Target API: Lightstreamer Web Client (Unified) v8.0.0

2019-10-18

Table of Contents

Chapter 1. Introduction

This document provides a conceptual introduction to the development of Lightstreamer Clients based on the Web Client Library.

A full JavaScript API reference is available online.

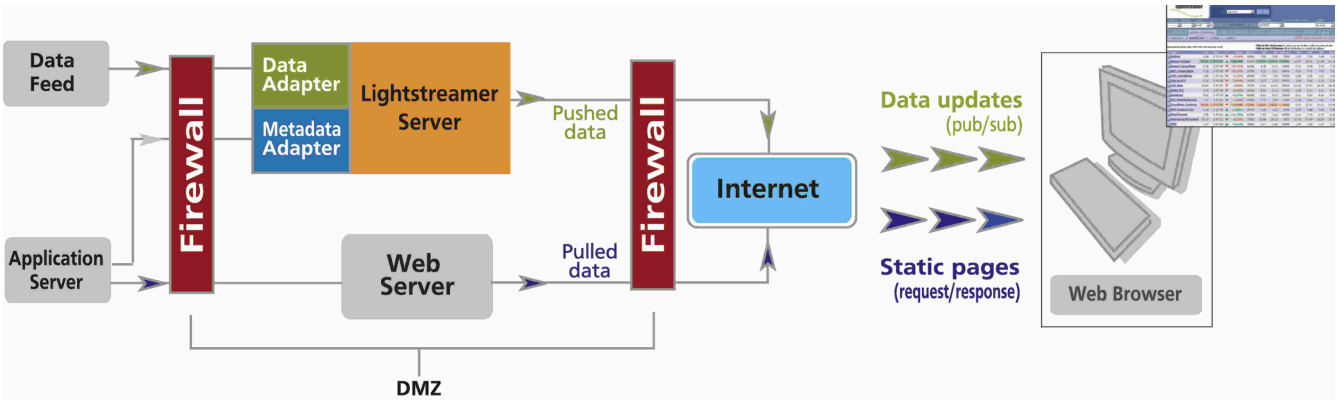
Chapter 2. Web Client Development

The **Lightstreamer Web Client API** is a set of **JavaScript** classes that can be included into any Web site, HTML5 application, or JavaScript application in general in order to make it "Lightstreamer-enabled", that is, ready to receive and send real-time data from/to Lightstreamer Server.

The Web Client Library supports all the major formats (UMD, CommonJS and ES modules) and bundlers (Webpack, Rollup and Browserify). Further information on how to integrate the library in a project is provided on the npm library page. It is also possible to customize the library to only include the desired classes by means of a build script distributed with the Github project.

2.1. Deployment Architecture

For demo purpose, Lightstreamer Server contains an internal web server that is able to serve static web resources. This makes it simple to distribute out-of-the-box demos (such as the Stock-List Demo) and to get started with Lightstreamer development. But for actual projects and production scenarios, it is highly recommended that an external web server is used (because Lightstreamer is optimized for pushing real-time data, not for serving static pages).



Typically, a Lightstreamer Server and any common web server (e.g. Apache, nginx, IIS or any other web server of your choice) are deployed in the DMZ network. Lightstreamer Server gets the data and metadata from the back-end servers that are protected by the second firewall. The web browser gets the static web pages, as usual, from your web server, then it connects to Lightstreamer Server to receive the flow of real-time updates. Of course, both servers can be clustered for fail-over and load balancing.

It is also possible to let a browser load the static web pages directly from the file system of the client machine, a case that is useful during development and testing.

2.2. Site/Application Architecture

Each page of the site that is going to receive and/or send real-time updates from/to Lightstreamer Server must include a LightstreamerClient instance, which transparently manages the connections to Lightstreamer Server.

The library is designed to work even outside of a browser context. This means you can use it within any JavaScript-based environment, including **PhoneGap**, **SmartTV**, and others. For the **Node.js**

environment, a similar, specialized library is made available as a different SDK (see the npm page for details).

2.3. The LightstreamerClient

An HTML page, produced through any kind of server technology (e.g. JSP, ASP, PHP, Node.js, etc.) can host a LightstreamerClient to connect to a Lightstreamer Server.

Below are the steps required to transform a normal HTML page into an HTML page receiving updates from a LightstreamerClient. (The code examples assume that the UMD variant of the library is deployed under the "/js" folder of the web server).

2.4. Basic Example

1. **Include the libraries** As a very first action, the HTML page should import, in its HEAD section, the **lightstreamer.js** JavaScript library.

   ```
   <script src="/js/lightstreamer.js"></script>
   ```

2. **Include the style sheets** All the **cascading style sheets (CSS)** declared in the Lightstreamer functions should be defined or linked in the HEAD section of the page.

3. **Create the LightstreamerClient object** To connect to a Lightstreamer Server, a LightstreamerClient object has to be created, configured, and instructed to connect to the Lightstreamer Server. A minimal version of the code that creates a LightstreamerClient and connects to the Lightstreamer Server on http://push.mycompany.com will look like this:

   ```
   var myClient = new LightstreamerClient("http://push.mycompany.com","MyAdapterSet");
   myClient.connect();
   ```

4. **Create a Subscription object** A simple Subscription containing three items and three fields to be subscribed in MERGE mode is easily created (see Lightstreamer General Concepts):

   ```
   var mySubscription = new Subscription("MERGE",["item1","item2","item3"],["field1",
   "field2","field3"]);
   mySubscription.setDataAdapter("MyDataAdapter");
   myClient.subscribe(mySubscription);
   ```

5. **Listen for real-time updates** Before sending the subscription to the server, usually at least one listener is attached to the Subscription instance in order to consume the real-time updates. The following code shows the value of the "field1" field in a JavaScript alert each time a new update is received for the subscription:

```
mySubscription.addListener({
    onItemUpdate: function(updateObject) {
        alert(updateObject.getValue("field1"));
    }
});
```

6. **The complete code** Below is the complete JavaScript code:

```
var myClient = new LightstreamerClient("http://push.mycompany.com", "MyAdapterSet
");
myClient.connect();
var mySubscription = new Subscription("MERGE",["item1","item2","item3"],["field1",
"field2","field3"]);
mySubscription.setDataAdapter("MyDataAdapter");
mySubscription.addListener({
    onItemUpdate: function(updateObject) {
        alert(updateObject.getValue("field1"));
    }
});
myClient.subscribe(mySubscription);
```

2.4.1. LightstreamerClient in a Web Worker

A LightstreamerClient instance can also be used inside a WebWorker.

1. **Create a Worker** To load the client in a Worker, a Worker is needed first. A listener on the worker can be used to receive messages from the worker itself. Simply put the following code in a HTML file

```
<script>
var worker = new Worker('index.js');
worker.addEventListener('message', function(event) {
    alert(event.data);
});
</script>
```

2. **Prepare the Worker Code** The *index.js* file specified in the above code will contain the JavaScript code used to connect to a Lightstreamer Sever. It starts including the needed library

```
importScripts("lightstreamer.js");
```

then the same code shown in the above example can be used with only one single difference, that is, a different approach is used to consume the Subscription data: such data is sent back as an event using the postMessage facility (note that the alert method is not even available in the Worker context):

```
mySubscription.addListener({
    onItemUpdate: function(updateObject) {
        postMessage(updateObject.getValue("field1"));
    }
});
```

2.5. Simple Widgets

The Lightstreamer Web Client Library includes some ready-made widgets, under the form of SubscriptionListener implementations that can be used in an HTML page to show the data received on a Subscription. You can use these widgets to get started or you may want to directly use any third-party library to display the data.

Below is a description of the two main implementations: StaticGrid and DynaGrid. Actually, such classes can also be used without listening on Subscription objects, but such usage is out of the scope of this document.

In the following description the term **cell** is used to refer to the atomic unity of visualization for data pushed by Lightstreamer and associated to a **field** (see General Concepts).

In order to declare a cell, a special **DIV** element or a special **SPAN** element should be used. It is possible to choose between a DIV and a SPAN tag for each cell, according to the front-end requirements.[1] Alternative ways for supplying push cells are available, which allow the use of any kind of tag.

2.5.1. StaticGrid

In a StaticGrid each cell is statically defined in the HTML page.

Each cell for this kind of grid can define the following special properties:

- [mandatory] **data-source**: this special property must be specified with its value being "lightstreamer" in order to authorize the StaticGrid to use such HTML element as a cell.
- [mandatory] **data-grid**: an identifier that is used to associate a cell with a StaticGrid instance: the same value has to be specified in the constructor of the StaticGrid.
- [mandatory] **data-row**: a number representing the row number this cell is associated with. As all the cells have to be manually defined, the number of rows in the grid is defined by the biggest value of this property in all the cells related to the StaticGrid (note that **data-item** can be used instead of data-row in some special cases).
- [mandatory] **data-field**: the name of the field to be associated with the cell (it can also be a field index)
- [optional] **data-num**: if there is the requirement to have more cells related to the same row/field pair, it is possible to specify this property in order to distinguish between such cells when handling the StaticGrid events.
- [optional] **data-update**: by default the content of a cell is updated with the received values. By

specifying this property, it is possible to target any property of the HTML element (e.g.: the src property of an IMG tag) or its stylesheet.

- [optional] **data-fieldtype** The StaticGrid class offers the *extractFieldList* and *extractCommandSecondLevelFieldList* functions to read from the html the data-field values of the associated cells. Using this property is possible to specify if a field has to be extracted during *extractFieldList* executions (missing data-fieldtype property or "first-level" value), during *extractCommandSecondLevelFieldList* (data-fieldtype set to "second-level") or neither (data-fieldtype set to "extra")

Example (HTML part)

```
<div data-source="lightstreamer" data-grid="quotes" data-row="1" data-field="field1">-
</div>
<div data-source="lightstreamer" data-grid="quotes" data-row="1" data-field="field2">-
</div>
<div data-source="lightstreamer" data-grid="quotes" data-row="2" data-field="field1">-
</div>
<div data-source="lightstreamer" data-grid="quotes" data-row="2" data-field="field2">-
</div>
```

Example (JavaScript part)

```
new StaticGrid("quotes",true);
```

Only the basic bits of the StaticGrid are described in this guide; check out the JSDoc for all the details.

2.5.2. DynaGrid

In a DynaGrid only one row of cells is defined in the HTML page (the **template**). New rows are then cloned from the given template row. The template row will be associated to the DynaGrid instance via its **id** property and will contain all of the necessary cells. The data-source property of this template has to be configured as if it was a cell.

Each cell for this kind of grid can define the following special properties:

- [mandatory] **data-source**: this special property must be specified with its value being "lightstreamer" in order to authorize the DynaGrid to use such HTML element as a cell.
- [mandatory] **data-field**: the name of the field to be associated with the cell (it can also be a field index)
- [optional] **data-num**: if there is the requirement to have more cells related to the same field, it is possible to specify this property in order to distinguish between such cells when handling the DynaGrid events.
- [optional] **data-update**: by default the content of a cell is updated with the received values. By specifying this property it is possible to target any property of the HTML element (e.g.: the src property of an IMG tag) or its stylesheet.

- [optional] **data-fieldtype**: the DynaGrid class offers the *extractFieldList* and *extractCommandSecondLevelFieldList* functions to read from the html the data-field values of the associated cells. Using this property is possible to specify if a field has to be extracted during *extractFieldList* executions (missing data-fieldtype property or "first-level" value), during *extractCommandSecondLevelFieldList* (data-fieldtype set to "second-level") or neither (data-fieldtype set to "extra")

Example (HTML part)

```
<div id="quotes" data-source="lightstreamer">
    <div data-source="lightstreamer" data-field="field1">-</div>
    <div data-source="lightstreamer" data-field="field2">-</div>
</div>
```

Example (JavaScript part)

```
new DynaGrid("quotes",true);
```

2.6. Troubleshooting

During development it is always possible to encounter issues preventing custom code from working as expected. If that happens you can follow the tips in this section to solve the issues.

2.6.1. Check for exceptions

The first thing to look at when facing an issue is the console of the browser in use. Most modern browsers offer some sort of built-in JavaScript console: fire it up and reload your application; if there is a piece of code throwing an exception it will be likely shown there.

2.6.2. Server-sent errors

When connecting to a server, subscribing to an item or sending a message, something can go wrong due to bad configurations, adapter errors or server constraints. In these cases the server will send back an error that will be exposed on the appropriate listener(s). You can register a listener on your objects to receive such notifications:

- **Connection errors** A ClientListener instance can be used to receive error notifications on LightstreamerClient instances. Note that the ClientListener interface has some methods other than the error one that we're going to see now.

```
var myLSClient = new LightstreamerClient();
myLSClient.addListener({
    onServerError: function(errorCode,errorMessage) {
        // here you can consume the error
    }
});
```

- **Subscription errors** A SubscriptionListener instance can be used to receive error notifications regarding a Subscription. Note that the SubscriptionListener interface has some methods other than the error ones that we're going to see now.

```
var mySubscription = new Subscription();
mySubscription.addListener({
    onSubscriptionError: function(errorCode,errorMessage) {
        //here you can consume the error
    },
    onCommandSecondLevelSubscriptionError: function(errorCode,errorMessage,relatedkey)
{
        //this one can only be fired in case a two-level subscription is created
        //here you can consume the error
    }
});
```

- **Message errors** A ClientMessageListener instance can be used to receive error notifications on sent messages. Note that the ClientMessageListener interface has some methods other than the error one that we're going to see now.

```
myLSClient.sendMessage(myMessage,mySequence,myTimeout,{
    onError: function(myMessage) {
        //here you can consume the error
    },
    onDeny: function(myMessage,denyCode,denyMessage) {
        //here you can consume the error
    }
});
```

2.6.3. Logging

The Lightstreamer Web Client API includes a simple LoggerProvider logging interface that can be implemented to consume log messages from within the library.

A ready-made implementation of such interface, together with several log appender classes, is distributed with the library. The configuration of this logging facility should be different between development and production environments.

The suggested approach is to use a ConsoleAppender configured at INFO level during development and a RemoteAppender configured at ERROR level or no appenders at all on production

deployments.

The log messages are also identified by a category.

- **Development setup** During development we suggest to setup a ConsoleAppender at INFO level to keep an eye on what is happening on the library. Most modern desktop browsers include (or can be extended with) a console showing the messages produced by such appender (for instance, look for the Developer Tools).

```
var loggerProvider = new SimpleLoggerProvider();
LightstreamerClient.setLoggerProvider(loggerProvider);
var myAppender = new ConsoleAppender("INFO","*");
loggerProvider.addLoggerAppender(myAppender);
```

- **Production setup** During production we suggest to remove the ConsoleAppender and add a RemoteAppender that will forward its messages to the configured Lightstreamer Server. To setup such logging the following code can be used (where myClient is the instance of LightstreamerClient connected to the server where the logging has to be sent):

```
var loggerProvider = new SimpleLoggerProvider();
LightstreamerClient.setLoggerProvider(loggerProvider);
var myAppender = new RemoteAppender("ERROR","*",myClient);
loggerProvider.addLoggerAppender(myAppender);
```

2.6.4. Still in trouble

If none of the above suggestions made you find a solution to your problems you can reach out to our public forums.

[1] See http://www.w3.org/TR/REC-html40/struct/global.html#h-7.5.4 for a formal specification of the difference between DIV and SPAN.