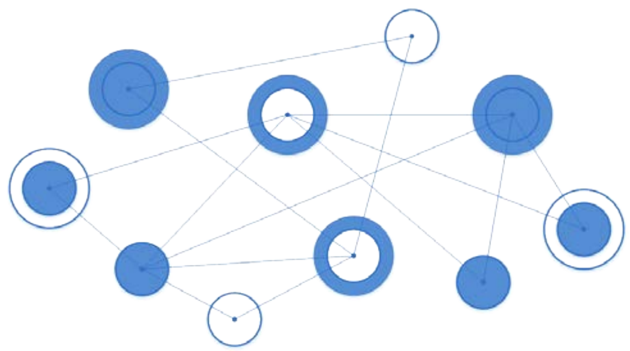


ConnectorJS

WebQueue

Developer Guide

Concepts and Reference

LANSA Group

Contents

ABOUT CONNECTORJS DOCUMENTATION **4**

CONNECTORJS WEBQUEUE DEVELOPER GUIDE 4
CONNECTORJS DOCUMENTATION LIBRARY 4
FINDING INFORMATION 5

SECTION 1: CONNECTORJS AND WEBQUEUE **6**

WHAT IS CONNECTORJS? 6
What is WebQueue? *6*
Use case suggestions *6*
WEBQUEUE CONCEPTS 7
Concept of operation *7*
Deployment options *8*
Benefits of using WebQueue *10*
Application examples using WebQueue *11*
WEBQUEUE Q&A 13

SECTION 2: USING WEBQUEUE **15**

CONFIGURATION 15
QUEUE NAME {QUEUEID} SYMBOLIC VARIABLE 16
RESPONSE OBJECT 16
AUTHENTICATION OPTIONS 17
STATUS AND ERROR MESSAGES 17
CLOSING CONNECTIONS 19
SEND AND RECEIVE MESSAGES 20
FILE TRANSFER USING WEBQUEUE 24
Uploading and downloading files *24*
Send files including various content types *27*
WEBSOCKET STATUS CODES 29

SECTION 3 - CODE SAMPLES **31**

CONDITIONS OF USE 31
Warranties *31*
Liabilities *31*
HOW TO USE THE CODE SAMPLES 31
Code samples *32*
VIEWING PDF DOCUMENTS IN A BROWSER 48

APPENDICES **52**

GLOSSARY 52
ASSUMED AND PREREQUISITE KNOWLEDGE 53
PROGRAM DATA QUEUE SIZE 54

Content © LANSA Group
All Rights Reserved

List of Figures

Figure 1: WebQueue Interaction with a Program Running on a Server 7
Figure 2: WebQueue Deployment Options 8
Figure 3: Configuration for Deployment Option 1 8
Figure 4: Deployment Option 2 Configuration Read Queue 9
Figure 5: Deployment Option 2 Configuration Update 9
Figure 6: Interactions between a Web Application and ConnectorJS Example 1 11
Figure 7: Interactions between a Web Application and ConnectorJS Example 2 12

List of Tables

Table 1: Benefits of using WebQueue 10
Table 2: WebQueue Q&A 13
Table 3: WebQueue Configurations 15
Table 4: Configuring the {QUEUEID} Symbolic Variable 16
Table 5: Response Object Properties 16
Table 6: WebQueue Message Type Definitions 17
Table 7: Sample Messages 18
Table 8: WebSocket Defined Status Codes 29
Table 9: WebSocket Reserved Status Codes 30
Table 10: WebSocket Private Status Codes 30
Table 11: Abbreviations and Terms 52
Table 12: Assumed and Prerequisite Knowledge 53

Document revision date: Wednesday, 14 June 2017

Content © LANSA Group
All Rights Reserved

About ConnectorJS documentation

ConnectorJS (CJS) is a JavaScript library containing APIs developers can call to use resources provided by IBM i servers. WebQueue is one of the ConnectorJS APIs. It uses the WebSocket protocol to bring IBM i resources and services to JavaScript in a web application.

ConnectorJS WebQueue Developer Guide

This guide describes the services WebQueue provides, and includes examples showing JavaScript code using the API.

Discover information about ConnectorJS features and services.	Section 1 – About ConnectorJS and WebQueue Learn about ConnectorJS and WebQueue and understand the role of ConnectorJS Server.
Use WebQueue API	Section 2 – Using WebQueue This section explains the JavaScript code and server configuration required for WebQueue.

ConnectorJS documentation library

ConnectorJS Administrator Guide	
ConnectorJS Concepts	
ConnectorJS Developer Reference	
ConnectorJS Installation Guide	
ConnectorJS WebQueue Developer Guide	
JSM HTTP Server Administrator Guide	ConnectorJS Server is a service managed by the JSM HTTP Server. The JSM HTTP Server Administrator Guide provides information about managing and configuring the JSM HTTP Server.

Content © LANSA Group
All Rights Reserved

Finding information

What do you want to do?	Consult these Resources
Find information about ConnectorJS.	ConnectorJS Concepts The guide provides an overview of ConnectorJS and explains the role of the ConnectorJS Server software.
Go to the WebQueue API code samples.	Read the code samples section in this guide.
Install ConnectorJS on a server.	ConnectorJS Installation Guide The guide explains how to install ConnectorJS on an IBM i server.
Learn out about administrator activities and responsibilities, fixing problems and managing day-to-day activities.	ConnectorJS Administrator Guide The guide explains administrative concepts, responsibilities, housekeeping and troubleshooting.
Set values for configuration items and parameters.	ConnectorJS Administrator Guide The guide explains how to configure ConnectorJS server software by setting values for the configuration items and parameters.
Troubleshoot problems	ConnectorJS Administrator Guide The guide includes a section describing troubleshooting options.
Understand how to use ConnectorJS IBM i APIs.	ConnectorJS Developer Reference The reference document explains API syntax to access IBM i resources, invoking APIs from JavaScript and includes code samples.
Understand how to use ConnectorJS WebQueue API.	Read this guide. The guide explains WebQueue API syntax, invoking the API from JavaScript and includes code samples.

Content © LANSA Group
All Rights Reserved

Section 1: ConnectorJS and WebQueue

This section provides conceptual information about ConnectorJS and the WebQueue API.

What is ConnectorJS?

ConnectorJS is a JavaScript library containing APIs developers can call to use resources provided by IBM i servers. The JavaScript APIs interface with ConnectorJS Server installed on a server. The product equips developers with interfaces they can use to access server resources (e.g. programs and commands) from JavaScript in web applications.

ConnectorJS Server	ConnectorJS Server is software installed on a server.
ConnectorJS JavaScript library	ConnectorJS JavaScript library provides an interface between the ConnectorJS Server software on a server and JavaScript written by developers.
HTML5, CSS and JavaScript	Developers build web applications that use APIs provided by ConnectorJS.

What is WebQueue?

WebQueue is an API provided by CJS that provides communication between a web application and a server via a WebSocket connection.

Using WebQueue, developers can send/receive messages to/from program data queues using JavaScript via a WebSocket connection. Programs monitoring the program data queues can access resources and services on an IBM i server.

Developers use JavaScript and other development tools when building web applications. At points in the applications where developers require server resources, they can use the WebQueue API to access the resources. WebQueue provides web applications with a full-duplex connection to a server, over which the application and the server can exchange messages.

The CJS JavaScript library interfaces with CJS Server to access requested resources. CJS Server invokes the resources to produce the required results, and returns responses to web applications.

Use case suggestions

Pushing data from a server to a browser

Developers can use WebQueue to push information to a web application from a server without waiting for a request from the web application.

Web applications can place messages on program data queues, or retrieve messages from program data queues. WebQueue can inform a web application when a message is waiting without the web application having to poll a server.

Viewing PDF documents in a browser

Developers can use WebQueue to invoke a server program that can generate a PDF document. When the document is ready for download, CJS Server can inform the web application by initiating a message and providing access to the document. The web application can download the PDF document and present it for viewing.

WebQueue concepts

Concept of operation

WebQueue provides WebSocket connections between web applications and CJS Server installed on an IBM i server. Connections are bi-directional which means that the server or the web application can initiate messages. Web applications can use a connection to interact with programs on a server in real-time. CJS Server uses program data queues to use as interfaces between web applications and server programs. Web applications can receive messages from the read data queue and send messages to the write data queue.

Figure 1 (page 7) illustrates a WebQueue interaction with a program running on a server.

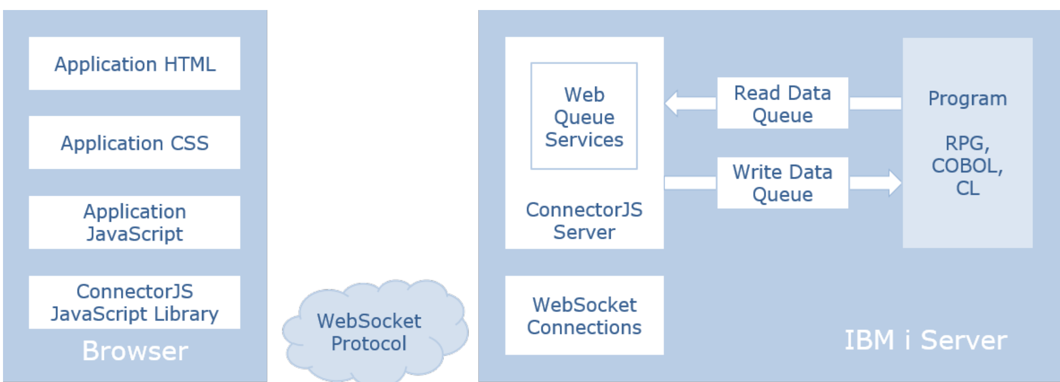


Figure 1: WebQueue Interaction with a Program Running on a Server

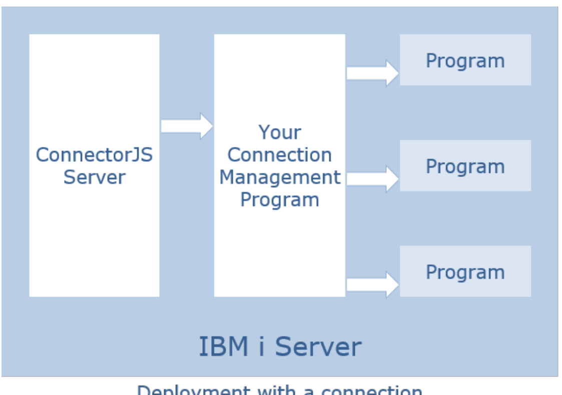
After a web application requests a connection to CJS Server, the browser and the server establish a WebSocket connection, CJS Server verifies the user, accepts the request, starts a job, creates a read program data queue, and an optional write program data queue. The data queues reside in a library defined in the CJS Server configuration file. Data queue names include a unique identification to prevent name collisions.

A web application can send messages to CJS Server and it forwards messages to the write data queue. Server programs monitoring the data queue can retrieve messages and act on the message content. A program running on a server can send messages to a web application via the read data queue. CJS Server forwards messages to a web application.

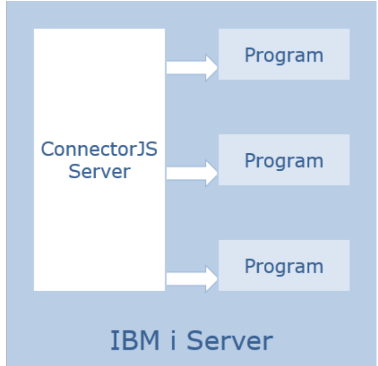
Content © LANSA Group
All Rights Reserved

Deployment options

Two deployment options are available (Figure 2, page 8). Option 1 uses a dedicated connection management program as an interface between CJS Server and programs on the server. Option 2 allows CJS Server to call programs on the server directly, without a connection management program.



Deployment with a connection management program.



Deployment without a connection management program.

Figure 2: WebQueue Deployment Options

Deployment option 1 employs a connection management program that CJS Server calls. Parameters in the configuration specify the connection program that CJS Server will call. The connection program chooses a program to run based on the content of the configuration file and parameters supplied by a web application, and starts the program, passing the library and names of the data queues. The significant feature of option 1 is the inclusion of business logic in the connection management program to determine the program that will process messages from web applications.

```
<parameter name: "service.queue.customer"             <!-- parameter definition name -->
  value= "{
    read: { object: 'crmlib/{QUEUEID}', length: 128, . . . },
    write: { object: 'crmlib/{QUEUEID}', length: 128, . . . },
    connect: { program: 'crmpgmlib/crmconnect,              <!-- connection program -->
        parameters: [ { name: 'QUEUE', length: 32 },
                      { name: 'QUEUEID', length: 6 },
                       . . . ]
            }" />
```

Figure 3: Configuration for Deployment Option 1

Figure 3 (page 8) shows a partial configuration where the parameter definition named CUSTOMER defines the connection management program as CRMCONNECT in the CRMPGMLIB library. Web applications define the customer object in an API call and the customer object configuration defines the program to call. Developers build the connection management program and insert business logic to decide which program(s) to call.

Option 2 avoids a connection management program. The configuration includes a parameter definition for each program that web applications can call. The applications must know and select the

Content © LANSA Group
All Rights Reserved

appropriate CJS configuration object definition. Option 2 places the business logic to determine a program to call in the web applications, complemented by properties and parameter definitions in the configuration file.

```
<parameter name: "service.queue.customerread"        <!-- parameter definition name -->
  value= "{
    read: { object: 'crmlib/{QUEUEID}', length: 128, . . . },
    write: { object: 'crmlib/{QUEUEID}', length: 128, . . . },
    connect: { program: 'crmpgmlib/custget,                  <!-- program to call -->
        parameters: [ { name: 'QUEUE', length: 32 },
                      { name: 'QUEUEID', length: 6 },
                       . . . ]
           }" />
```

Figure 4: Deployment Option 2 Configuration Read Queue

Figure 4 (page 9) shows a partial configuration with a parameter definition named CUSTOMERREAD.

```
<parameter name: "service.queue.customerupdate"      <!-- parameter definition name -->
  value= "{
    read: { object: 'crmlib/{QUEUEID}', length: 128, . . . },
    write: { object: 'crmlib/{QUEUEID}', length: 128, . . . },
    connected: { program: 'crmpgmlib/custmaint,              <!-- program to call -->
        parameters: [ { name: 'QUEUE', length: 32 },
                     { name: 'QUEUEID', length: 6 },
                       . . . ]
           }" />
```

Figure 5: Deployment Option 2 Configuration Update

Figure 5 (page 9) shows a partial configuration with a parameter definition named CUSTOMERUPDATE.

Web applications select the customerread definition to retrieve customer data. CJS Server uses data from the CUSTOMERREAD definition to start the CUSTGET program and it returns the customer data by placing message(s) on the read data queue. Web applications select the CUSTOMERUPDATE definition to insert or update customer data. ConnectorJS uses data from the CUSTOMERUPDATE definition to start the CUSTMAINT program.

Content © LANSA Group
All Rights Reserved

Benefits of using WebQueue

Table 1 (page 10) describes some of the benefits of WebQueue.

Table 1: Benefits of using WebQueue

Features	What WebQueue can Do
Any IBM i program can participate in a browser conversation	IBM i programs that can send and receive messages via a program data queue can interact with a web application running in a browser.
Asynchronous processing	A web application can issue a request for a report. A server program can create the report and send a message to the web application when the report is ready. Meanwhile, the web application is free (i.e. no blocking) to perform other tasks.
Develop well performing applications	Developers can build applications that perform well without a detailed knowledge of application performance optimisation.
Fast and efficient	WebSocket communication is faster than HTTP request/response mode of operation.
Full duplex	Communications between a web application and a server operates in both directions.
Hide the complexity of socket communications	Developers can use WebQueue without understanding HTTP or socket communications.
Loose coupling between server and client	WebQueue's message architecture simplifies web application and server conversations. They exchange messages, initiated by either party, without dependencies. For example, when a web application issues a request it requires no knowledge of actions that will occur on the server. Similarly, server processes require no knowledge of the web applications.
Low CPU usage	Server CPU usage is low when idle because of the IBM i wait state.
Minimum resources required when idle.	Uses minimum server resources when idle. Server application design influences the resource use.
Modular application design	Applications use a modular design because of the separation of the user interface from server processes.
No HTTP/CGI API interface	WebQueue operates without the CGI API.
No session tokens	Web application and server conversations require no session tokens to manage state. Communication is real-time over a connection that remains available until the web application or CJS Server closes the session and the connection.

Features	What WebQueue can Do
Permanent connection	Connections remain permanent until closed by a web application, server, or timeout.
Push	Server programs can send messages to web applications without waiting for a request from a web application.
Same-origin policy	No same-origin policy constraints apply.
Server applications run as background tasks	Developers can build server applications using Command Language (CL), RPG, and LANSA as if they are batch programs using data queues. No user interface processing required.
Simple architecture	WebQueue requires no complex architecture to handle HTTP/XHR and AJAX requests and responses.
Simple to use with LANSA BIF's WEBQ_SEND and WEBQ_RECEIVE	LANSA BIF's include WEBQ_SEND and WEBQ_RECEIVE for use by LANSA developers.

Application examples using WebQueue

The following examples illustrate interactions between a web application, CJS and a server program. The methods behind the interactions include requests, messages and submit job. The examples assume an operational WebSocket connection between a browser and CJS on a server.

Example 1

The requirements are to run a program on a server and receive a message when the program terminates. The server program will gather data, prepare a report, generate a PDF copy of the report, and inform the web application on completion, sending the document name and location in a message.

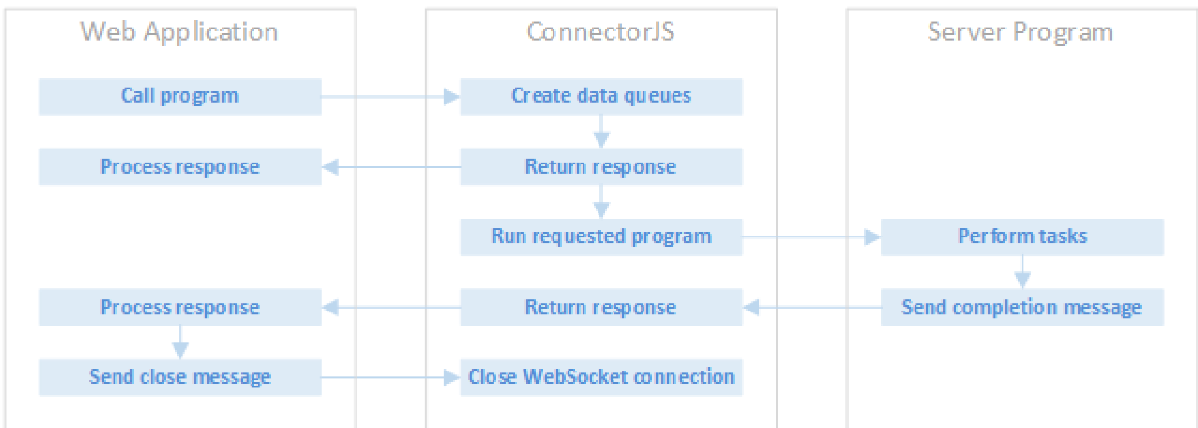


Figure 6: Interactions between a Web Application and ConnectorJS Example 1

Content © LANSA Group
All Rights Reserved

Figure 6 (page 11) shows the request and response sequence between a web application, CJS Server and a server program. The web application runs a server program via WebQueue and parameters in the API and the CJS configuration identify the program. CJS Server creates read and write data queues, and sends a response (including library and data queue names) to the web application. Then CJS Server starts the server program, passing the names and libraries of the data queues. The server program runs to completion and sends a completion message to the read data queue. CJS Server takes the message from the data queue, prepares a response and sends the response to the web application. The web application sends a close message to CJS Server and it closes the WebSocket connection.

Suppose the program running on the server requires several long running steps. The server program can send progress messages at the completion of each step to inform the web application. This action will generate additional messages in the read data queue. The web application can process the progress messages and display a progress bar.

Example 2

The requirements are to run a program on a server, exchange request and response messages between a web application and a server program to retrieve customer data from a database. The web application will send information to identify the customer. The server program will retrieve customer data from a database, send it to the web application and close the WebSocket connection.

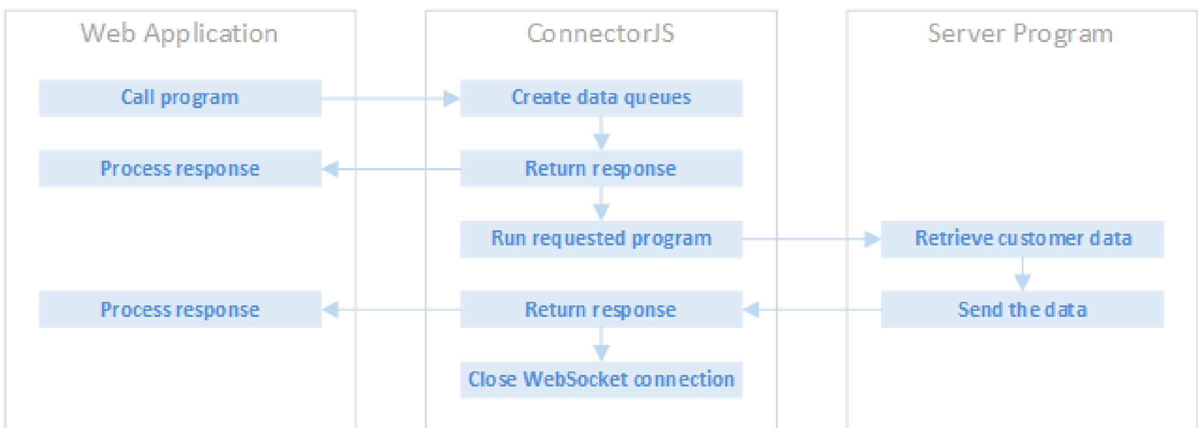


Figure 7: Interactions between a Web Application and ConnectorJS Example 2

Figure 7 (page 12) shows the request and response sequence between a web application, CJS Server and a customer information program on a server. The web application runs the customer information program via WebQueue. CJS Server creates a read data queue and a write data queue, and sends a response (including library and data queue names) to the web application. Then CJS Server starts the server program, passing the names and libraries of the data queues. The customer information program retrieves the customer data from a database and sends a message to the read data queue (if the customer data is voluminous the program may send multiple messages). CJS Server takes the message from the data queue, prepares a response and sends the response to the web application. Then CJS closes the WebSocket connection.

Content © LANSA Group
All Rights Reserved

WebQueue Q&A

Table 2 (page 13) presents questions and answers about configuring and managing WebQueue.

Table 2: WebQueue Q&A

Questions	Answers
Can I choose the library for CJS data queues?	Yes. The configuration of a WebQueue object can define a library name.
Can I define names for CJS data queues?	ConnectorJS manages data queue names. The maximum length of a data queue name is 10 characters. {QUEUEID} uses 6 characters allocated by CJS. Developers can use the remaining 4 characters as a prefix or suffix to {QUEUEID}.
Can I send or receive large files using the WebQueue API?	While it is possible to send and receive large files, this is not a recommended practice.
How does a Web application know about messages in the read data queue?	CJS Server retrieves messages from the read data queue and sends the message content to a web application via a call-back function (e.g. onMessage).
How many data queues will CJS Server create?	CJS Server creates a read data queue and optional write data queue for each connection.
Maximum file size for uploads	The suggested maximum size is 150 megabytes.
WebSocket jobs	Each WebSocket connection has a server job in the QUSRWRK subsystem.
What controls the number of data queues?	The configuration of a WebQueue object defines the data queues. A read data queue is mandatory and a write data queue is optional.
What does ConnectorJS mean?	ConnectorJS is the product name.
What does LUICONNECTOR mean?	LUICONNECTOR is an internal object naming convention used by CJS.
What happens to the data queues after a connection closes?	CJS Server will remove the data queues associated with the connection.

Questions	Answers
What is {QUEUEID}?	{QUEUEID} is a configuration variable. When ConnectorJS creates a data queue it assigns a unique identifier and uses the identifier as part of the data queue name. The configuration object defines a naming convention. ConnectorJS substitutes the job number of the server job into the {QUEUEID} pattern. The job number is zero padded to 6 digits in length.
What will happen if I configure a data queue name without the identifier?	Not recommended CJS Server assumes each connection has exclusive use of its data queues. Naming data queues without the identifier will create a read data queue, and optionally, a write data queue. The first connection will create the data queues. Subsequent connections will fail when attempting to create the data queues (as the data queues exist). The consequence is only one connection can operate successfully at a time.

Content © LANSA Group
All Rights Reserved

Section 2: Using WebQueue

WebQueue requires data defining server resources and the IBM i server (host). To use the API, developers code the API interface using JavaScript and deploy the CJS JavaScript library with the Web application JavaScript.

Configuration

The configuration file includes connector tags that define connector APIs and wrap the parameters associated with a connector. The structure is a connector element.

Configuration
<connector name= "webqueue" Parameters associated with the webqueue connector </connector>

The connector name is webqueue.

Connectors may include multiple parameter directives.

Table 3 (page 15) shows the structure of the configuration for the WebQueue APIs.

Table 3: WebQueue Configurations

Configuration Examples
<match uri="/service/webqueue.jsp" class="com.lansa.mobile.service.HTTPServiceWebQueue"> . . . . <host> . . . . </host> </match>

CJS requires explicit configuration to use server resources. Directives in the configuration file define parameters associated with the API.

IBM i object authority takes precedence over directives authorising use of server resources.

The configuration section of the ConnectorJS Administration Guide explains how to configure CJS Server and includes examples of directives required for WebQueue.

Queue name {QUEUEID} symbolic variable

The {QUEUEID} symbolic variable represents part or all of the name of read and write data queues created by CJS. The data queues provide the messaging interface used to send and receive messages between a server and a web application running in a browser, across a WebSocket connection.

Each WebSocket connection has a server job in the QUSRWRK subsystem. CJS Server zero pads the job number to a length of 6 digits to derive a value for the {QUEUEID} symbolic variable.

Message queue name maximum length is 10 characters. CJS uses 6 characters as the message queue name, represented in the configuration as {QUEUEID}. The remaining 4 characters are available for developers. The 4 characters can be a prefix or suffix to {QUEUEID} and are optional.

Table 4 (page 16) illustrates data queue names with their configuration.

Table 4: Configuring the {QUEUEID} Symbolic Variable

Configuration	Queue Name	Explanation
None	438712	CJS creates queue names as 6-digit numbers when the configuration includes no specific naming directives.
RPT{QUEUEID}R	RPT567321R	The queue name includes RPT as a prefix, the 6-digit number, and R as a suffix.
{QUEUEID}ACC	123456ACC	The queue name includes the 6-digit number and ACC as a suffix.
W{QUEUEID}	W729469	The queue name includes W as a prefix and the 6-digit number.

CJS Server returns the job number to the web application as the queue identification property.

Response object

The response object is where WebQueue returns values and/or describe the success or failure of an API call. Typical JavaScript code for the response object is as follows.

```
vary response = LUICONNECTOR.execute (host, service, null);
```

The response is a JSON object.

Table 5 (page 16) describes the response object.

Table 5: Response Object Properties

Object	Properties	Definition
Response	Response type	Response type values are status or error.

Content © LANSA Group
All Rights Reserved

Object	Properties	Definition
Response type error	Message	The message property contains text that describes the nature of the error.
Response type status	Status	The response type status values are OK or NOTEXIST.

Use the JavaScript JSON.stringify (response) to examine the response object.

Authentication options

When creating WebSocket connections web applications must provide user credentials. Web applications can provide credentials in two ways.

- User profile (identification) and password
- Symbolic name

The API call may include credentials in the form of a user profile (identification) and a password. In this case, web applications must collect the credentials, e.g. via a user login form. Use this authentication method when it is important to keep track of individual users on the server.

A second method is a symbolic name representing a user defined in the configuration. A web application will provide the symbolic name on the API call and CJS Server will validate the name against the configuration, and if valid, login using the user profile associated with the symbolic name. A symbolic name is a string of characters prefixed by an asterisk, *FINANCE is an example. Use this authentication method when you want to use a generic user on the server.

Status and error messages

WebQueue provides information about the success of API calls by sending status and/or error messages to a web application. Status messages describe events or provide operational information. Error messages include information about failures in the API call. Developers can retrieve message content via a call back function.

Status and error messages are JSON objects and that include information about request status or errors. Developers can use ConnectorJS tracing services to collect information about errors.

Table 6 (page 17) defines the message types.

Table 6: WebQueue Message Type Definitions

Type	Definitions
Accept	CJS Server has accepted a message sent from a web application.
Close	The WebSocket connection is closed. The web application or CJS Server closed the connection.

Type	Definitions
Connected	The connection is active.
Error	An error occurred when processing a request.
Open	A WebSocket connection between a web application and CJS Server is open but not ready to accept messages.
Push	CJS Server has pushed (sent) a message to a web application with optional binary data.
Ready	CJS Server is ready to send and receive messages.
Reject	CJS Server rejected a message sent from a web application. Typically, reject messages arise when a request contravenes a configuration property or parameter value. For example, suppose message length in the configuration is 1024, CJS Server will reject messages whose length exceeds 1024.

Table 7 (page 18) provides examples of messages.

Table 7: Sample Messages

Type	JSON Format Message Content
Accept	{"sversion": "1.2.1", "status": "ACCEPT", "type": "status", "corelationid": 27} {"sversion": "1.2.1", "status": "ACCEPT", "type": "status", "corelationid": "XYZ"} {"sversion": "1.2.1", "status": "ACCEPT", "type": "status", "corelationid": {"id":495}}
Close	{"type": "status", "status": "CLOSE", "clean": true, "code": 4500, "reason": "Normal termination"}
Open	{"type": "status, "status": "OPEN"}
Push	{"messageid": {"qid": "040576", "tid": 1434949403632, "cid": "1"}, "sversion": "1.2.1", "value": "Reports start", "status": "PUSH", "sender": "", "type": "status"} {"messageid": {"qid": "040576", "tid": 1434949403661, "cid": "2"}, "sversion": "1.2.1", "value": "Reports complete", "status": "PUSH", "sender": "", "type": "status"}
Ready	{"sversion": "1.2.1", "qid": "040576", "queue": "REPORTS", "status": "READY", "type": "status"}

Type	JSON Format Message Content
Reject	{"reason": "Data exceeds maximum length", "sversion": "1.2.1", "code": 104, "status": "REJECT", "type": "status", "corelationid": "LONG"}

Closing connections

Web applications can close a connection to a server by sending a close message to CJS.

The server can close a connection. A program running on a server can send a close message to a CJS data queue. Close messages are a JSON string value, for example:

{ type: "action", action: "close", code: 4123, reason: "Close connection normal termination" }

Code 4123 is a private WebSocket status code and can be any number in the range 4000 to 4999. The reason text, "Close connection normal termination", is an optional explanation for closing the connection.

The CJS server interprets the message, sends a close message to the Web application, and closes the connection.

This example Command Language (CL) program sends a server-originated close connection message.

Server Originated Close Connection Example Command Language Program

```
PGM

DCL VAR(&SNDLIB) TYPE(*CHAR) LEN(10)
DCL VAR(&SNDNME) TYPE(*CHAR) LEN(10)
DCL VAR(&SNDLEN) TYPE(*DEC)  LEN(5 0)
DCL VAR(&SNDDTA) TYPE(*CHAR) LEN(100)
DCL VAR(&KEYLEN) TYPE(*DEC)  LEN(3 0)
DCL VAR(&KEYDTA) TYPE(*CHAR) LEN(40)

CHGVAR VAR(&SNDLIB) VALUE('TSTLIB')
CHGVAR VAR(&SNDNME) VALUE('TSTSND')
CHGVAR VAR(&SNDDTA) +
       VALUE('{type:"action", action: "close", +
                                code:4123, reason: "My reason for closing"}')
CHGVAR VAR(&SNDLEN) VALUE(100)
CHGVAR VAR(&KEYDTA) VALUE('MYKEY')
CHGVAR VAR(&KEYLEN) VALUE(40)

CALL PGM(QSNDDTAQ) PARM(&SNDNME &SNDLIB &SNDLEN &SNDDTA &KEYLEN &KEYDTA)

ENDPGM
```

Send and receive messages

Definitions and syntax

Syntax	Create connection
	    | connection | = LUICONNECTOR.createWebQueue () ;
	Open connection
	    response = | connection |.open
	Close connection
	    response = | connection |.close()
	Close connection with reason
	    response = | connection |.close(| close reason |)
	Create message content
	    message = { | message content | }
	Send message to ConnectorJS

	response = | connection |.send (message) Receive message from ConnectorJS Function onMessage (message) Receive message and binary object from ConnectorJS Function onMessage (message, data)
	The connection object defines a WebSocket connection with CJS Server. The response object will receive acknowledgements from CJS Server. The message object contains the data sent in a message, and | message data | represents the actual data, e.g. text or a JSON object. Developers can choose names for these objects.
Close reason	The close reason comprises a reason code (4000 to 4999) and a text string. Developers can issue a close without a reason. The WebSocket specification allocates the reason code range 4000 to 4999 for use by applications. Reason codes from 0 to 3999 and above 4999 are reserved.
Call back function	Developers can choose a name for the call back function. The examples use onMessage as the name of the call back function.
Host	Host defines the server resource URI (endpoint) and a user profile. vary host = { endpoint: "| resource URI |", profile: "| user profile |" } ; Example using John D's user profile and password. Host = { endpoint: "ws://chicago:6563/service/webqueue.jsp", profile: "JOHND", password: "password" } ; Example using a profile defined in the configuration host = { endpoint: "ws://chicago:6563/service/webqueue.jsp", profile:"*PRD" } ; The generic value *PRD points to a configuration directive that defines the user profile.

Code examples

Code Examples	Send and Receive Messages

Create and open a connection	`var m_queue = null ;`
	`function queueOpen ()`
	`{`
	`m_queue = LUICONNECTOR.createWebQueue () ;`
	`var host = { endpoint: "ws://chicago:6563/service/connector.jsp",`
	`profile: "JOHND", password: "password" } ;`
	`var response = m_queue.open`
	`( host, {queue:"REPORTS",key:"Day286"}, onMessage ) ;`
	`}`

Close a connection with a reason	`function queueClose ()`
	`{`
	`var response = m_queue.close (4510, "Abnormal termination" ) ;`
	`}`

Close a connection without a reason	`function queueClose ()`
	`{`
	`var response = m_queue.close () ;`
	`}`

Send a message to a data queue via a web queue connection	`function queueSend ()`
	`{`
	`var message = {Customer: "Smith"} ;`
	`var response = m_queue.send ( message ) ;`
	`}`

Receive a message from an IBM i server	`function onMessage ( message )`
	`{`
	Insert code to process the message.
	`}`

When WebQueue recognizes a close object (JSON), it sends a message to the client and closes the connection.

```
{type: "action", action: "close", code: 4123, reason: "| reason for close |"}
```

The syntax of the close object includes a reason code and optional text explaining the reason for closing a connection.

The configuration items required for this service are as follows.

Configuration

```
<match uri="/service/websocket.jsp"
   class="com.lansa.mobile.service.HTTPServiceWebSocket" trace="true" clienttrace="false">
   <parameter name="service.default.host" value="LOCAL"/>
   <parameter name="service.remote.activation" value=""/>
   <parameter name="service.origin" value="*"/>
   <parameter name="service.origin" value="http://chicagoadmin:1099"/>

   <host name="LOCAL" system="LOCALHOST">
      <parameter name="service.access.log" value="www/instance/logs/websocket.log"/>
      <parameter name="service.user.deny" value="NATHANC"/>
      <parameter name="service.user.allow" value="*USER"/>

      <parameter name="service.queue.REPORTS"
         value="{
           limit: 10485760
           read: {object: 'ERPLIB/RPT{QUEUEID}R', wait: 30, reclaim: true, ssi: true, length: 128 },
           write:{object: 'ERPLIB/RPT{QUEUEID}W', length:128 },
           upload: {folder:'/reports/{QUEUEID}'},
           connect: {
              program: 'ERPPGMLIB/ERPREPORTS',
              parameters: [ { name: 'QUEUE', length:32 },
                            { name: 'QUEUEID', length:6 },
                            { name: 'QUEUEUSER', length:10 },
                            { name: 'QUEUERLIB', length:10 },
                            { name: 'QUEUERNME', length:10 },
                            { name: 'QUEUEWLIB', length:10 },
                            { name: 'QUEUEWNME', length:10 },
                            { name: 'USERPARAM1', length:20 } ]
           }
      }" />
   </host>
</match>
```

File transfer using WebQueue

Uploading and downloading files

WebQueue can upload files from a web application to a server and download files to a web application from a server. The WebQueue architecture supports uploading (sending) and downloading (receiving) small files.

Developers should not use WebQueue as a general-purpose file transfer service. To download large files, send a message including the file location to a web application using WebQueue. JavaScript code in the web application can process the message and download the file using HTTP Get. Use a program on the server to decide whether a file is small or large.

Syntax and definitions

<table>
<tr><td rowspan="2">Syntax</td><td>Create connection
    | connection | = LUICONNECTOR.createWebQueue () ;
Open connection
    response = | connection |.open
Close connection
    response = | connection |.close()
Close connection with reason
    response = | connection |.close(| close reason |)
Upload file via ConnectorJS
| connection |.send (
    "| message identifier |",
      { type: "action",
        action: "sendfile",
        name: | file object name property| ,
        mimetype: "| mime type |", value: "| optional data, comments |"},
    | file object |) ;
Receive files from ConnectorJS
    Function onMessage (message, data)</td></tr>
<tr><td>The connection object defines a WebSocket connection with ConnectorJS.
The response object will receive acknowledgements from ConnectorJS.
The file object represents the file.
Developers can choose names for these objects.</td></tr>
<tr><td>Send object</td><td>The send object is a JSON formatted object.</td></tr>
</table>

All Rights Reserved

	Developers can assign a message identifier and use it to track related messages. Type is always action. Action is send when uploading files. Name is the file name, i.e. the name property of a file object. Mime type defines the type of file (e.g. application/octet-stream or pdf). Customer data
Close reason	The close reason comprises a reason code (4000 to 4999) and a text string. Developers can issue a close without a reason. The WebSocket specification allocates the reason code range 4000 to 4999 for use by applications. Reason codes from 0 to 3999 and above 4999 are reserved.
Call back function	Developers can choose a name for the call back function. The examples use onMessage as the name of the call back function.
Host	Host defines the server resource URI (endpoint) and a user profile. Var host = { endpoint: "| resource URI |", profile: "| user profile |" } ; Example using John D's user profile and password. Host = { endpoint: "ws://chicago:6563/service/connector.jsp ", profile: "JOHND", password: "password" } ; Example using a profile defined in the configuration host = { endpoint: "ws://chicago:6563/service/connector.jsp ", profile:"*PRD" } ; The generic value *PRD points to a configuration directive that defines the user profile.

Code examples

Code Examples	Upload and Download Files

Create and open a connection	`var m_queue = null ;`
	`function queueOpen ()`
	`{`
	`m_queue = LUICONNECTOR.createWebQueue () ;`
	`var host = { endpoint: "ws://chicago:6563/service/connector.jsp ",`
	`profile: "JOHND", password: "password" } ;`
	`var response = m_queue.open`
	`( host, {queue: "filetransfer" }, onMessage ) ;`
	`}`

Close a connection with a reason	`function queueClose ()`
	`{`
	`var response = m_queue.close (4560, "File transfer completed" ) ;`
	`}`

Close a connection without a reason	`function queueClose ()`
	`{`
	`var response = m_queue.close () ;`
	`}`

Receive a message from an IBM i server	`function onMessage ( message )`
	`{`
	`Insert code to process the message.`
	`}`

Code Examples	Upload and Download Files

Upload a file to a folder on a server	function queueSend ()
	{
	var file = document.getElementById ('id-file').files[0] ;
	if (file instanceof File)
	{
	m_queue.send ("DE47",
	{type: "action",
	action: "sendfile",
	name: file.name,
	mimetype: "application/octet-stream",
	value: "| optional data, comments |"},
	file) ;
	}
	}

Send files including various content types

Developers can use WebQueue to send files to a web application. The files can contain different content including HTML, JavaScript, text, images or PDF documents. The web application can add the HTML content to a page, execute the JavaScript, include an image on a page or open a PDF document viewer.

The following Command Language (CL) sample server program sends five files to a web application.

```
CHGVAR VAR(&DTA) +
VALUE('{type: "action", action: "sendfile",+
path: "/devjsm/instance/www/instance/htdocs/ConnectorJS/MyText.txt",+
name: "abc.txt", mimetype: "text/plain"}')
CALLSUBR SUBR(SEND)

CHGVAR VAR(&DTA) +
VALUE('{type: "action", action: "sendfile",+
path: "/devjsm/instance/www/instance/htdocs/ConnectorJS/MyJS.txt",+
name: "abc.txt", mimetype: "text/javascript"}')
CALLSUBR  SUBR(SEND)
```

```
CHGVAR VAR(&DTA) +
VALUE('{type: "action", action: "sendfile",+
path: "/devjsm/instance/www/instance/htdocs/ConnectorJS/MyHTML.txt",+
name: "abc.txt", mimetype: "text/html"}')
CALLSUBR  SUBR(SEND)

CHGVAR VAR(&DTA) +
VALUE('{type: "action", action: "sendfile",+
path: "/devjsm/instance/www/instance/htdocs/ConnectorJS/MyImage.jpg",+
name: "abc.jpg", mimetype: "image/jpeg"}')
CALLSUBR  SUBR(SEND)

CHGVAR VAR(&DTA) +
VALUE('{type: "action", action: "sendfile",+
path: "/devjsm/instance/www/instance/htdocs/ConnectorJS/MyPDF.pdf",+
name: "abc.pdf", mimetype: "application/pdf"}')
CALLSUBR  SUBR(SEND)
```

The code is a test program to illustrate sending files to a web application. The send subroutine sends a message to a CJS program data queue.

WebSocket status codes

The WebSocket protocol defines status codes as defined, reserved or private.

Use only private status codes to extend the defined status codes for your Web applications.

For information about WebSocket see, the Internet Engineering Task Force (IETF) specification for RFC 6455, the WebSocket Protocol, December 2011, at http://www.rfc-editor.org/info/rfc6455.

Table 8 (page 29) presents the defined status codes.

Table 8: WebSocket Defined Status Codes

Codes	Definition
1000	Indicates a normal closure, meaning that the purpose for which the connection was established has been fulfilled.
1001	Indicates that an endpoint is "going away", such as a server going down or a browser having navigated away from a page.
1002	Indicates that an endpoint is terminating the connection due to a protocol error.
1003	Indicates that an endpoint is terminating the connection because it has received a type of data it cannot accept (e.g., an endpoint that understands only text data MAY send this if it receives a binary message).
1004	Reserved. The specific meaning might be defined in the future.
1005	This is a reserved value and MUST NOT be set as a status code in a Close control frame by an end point. It is designated for use in applications expecting a status code to indicate that no status code was actually present.
1006	This is a reserved value and MUST NOT be set as a status code in a Close control frame by an end point. It is designated for use in applications expecting a status code to indicate that the connection was closed abnormally, e.g., without sending or receiving a Close control frame.
1007	Indicates that an endpoint is terminating the connection because it has received data within a message that was not consistent with the type of the message (e.g., non-UTF-8 [RFC3629] data within a text message).
1008	Indicates that an endpoint is terminating the connection because it has received a message that violates its policy. This is a generic status code that can be returned when there is no other more suitable status code (e.g., 1003 or 1009) or if there is a need to hide specific details about the policy.
1009	Indicates that an endpoint is terminating the connection because it has received a message that is too big for it to process.

Content © LANSA Group
All Rights Reserved

Codes	Definition
1010	Indicates that an endpoint (client) is terminating the connection because it has expected the server to negotiate one or more extension, but the server didn't return them in the response message of the WebSocket handshake. The list of extensions that are needed SHOULD appear in the /reason/ part of the Close frame. Note that the server does not use this status code, because it can fail the WebSocket handshake instead.
1011	Indicates that a server is terminating the connection because it encountered an unexpected condition that prevented it from fulfilling the request.
1015	This is a reserved value and MUST NOT be set as a status code in a Close control frame by an end point. It is designated for use in applications expecting a status code to indicate that the connection was closed due to a failure to perform a TLS handshake (e.g., the server certificate can't be verified).

Table 9 (page 30) presents the reserved status codes.

Table 9: WebSocket Reserved Status Codes

Codes	Definition
0-999	Status codes in the range 0-999 are not used.
1000-2999	The WebSocket protocol reserves status codes in the range 1000-2999 for definition by the protocol, its future revisions, and extensions specified in a permanent and readily available public specification.
3000-3999	Libraries, frameworks, and applications reserve status codes in the range 3000-3999 and registered directly with IANA. The protocol does not interpret these codes.

Table 10 (page 30) presents the private status codes.

Table 10: WebSocket Private Status Codes

Codes	Definition
4000-4999	Status codes in the range 4000-4999 are for private use and thus can't be registered. Such codes can be used by prior agreements between WebSocket applications. The protocol does not interpret these codes.

Content © LANSA Group
All Rights Reserved

Section 3 - Code samples

This section includes HTML page, JavaScript and IBM i command language program to test the WebQueue API.

Conditions of use

You can copy, amend and use the CJS JavaScript and HTML code samples under the warranty and liability conditions.

Warranties

The ConnectorJS JavaScript and HTML code samples are provided "as is". The LICENSOR makes no representation or warranties, express or implied, with respect to the code samples, including without limitation warranties of fitness for a particular use or purpose, merchantability, non-infringement, or that the ConnectorJS JavaScript and HTML code samples will operate without interruption or be free from errors, and LICENSOR hereby disclaims all such representations and warranties.

The ConnectorJS JavaScript and HTML code samples may not be at the level of performance, compatibility or safety of generally available LICENSOR software products. You shall have sole responsibility for adequate protection and backup of data or equipment used in connection with the ConnectorJS JavaScript and HTML code samples and You shall not claim against LICENSOR for lost data, re-run time, inaccurate input, work delays or lost profits resulting from the use of the JavaScript and HTML code samples.

Liabilities

The LICENSOR shall not be liable for any damages, whether special, direct, indirect, incidental, consequential or other damages, including without limitation, damages for loss of business profits, business interruption, loss of business information or any other pecuniary law under any theory of liability (including tort contract, or any other theory) whether suffered by You or any other user of the ConnectorJS JavaScript and HTML code samples, or any third party, even if LICENSOR was advised of the possibility of such damages.

How to use the code samples

The JavaScript code samples require modification to run successfully in your context. For example, the server (endpoint or host) and user profile in the samples will not match your server and users.

The JavaScript code samples include the JavaScript, HTML, configuration parameters and source code for programs to run on the server. A typical code structure defines the endpoint, calls the API, processes the response, and includes error handling.

Content © LANSA Group
All Rights Reserved

The samples include HTML, JavaScript and IBM i command language source code (where applicable). The JavaScript can go into the script tag in the head section of the HTML or save it as a separate file and add a reference in the head section.

Code samples

HTML page to test the WebQueue API.

WebQueue HTML Test Page

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>ConnectorAPI WebQueue Example</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <script type="text/javascript" src="/axes/lui-encrypt-min.js"></script>
    <script type="text/javascript" src="/axes/lui-connector-min.js"></script>
    <script type="text/javascript">
      <!-- JavaScript code goes here or a reference to JavaScript in a file  →
    </script>
  </head>
  <body>
    <button onclick="queueOpen()">Open</button>
    <button onclick="queueSend()">Send</button>
    <button onclick="queueClose()">Close</button>
    <input id="id-file" type="file"/>
    <p></p>
    <img id="id-image" width="300" height="300"/>
    <p></p>
    <iframe id="id-frame" src="/JavaScriptAPI/MyFiller.pdf"
         width="100%" height="25%"></iframe>
    <p></p>
    <a id="id-save" target="_blank"></a>
    <p></p>
    <object id="id-object" data="/JavaScriptAPI/MyFiller.pdf"
          type="application/pdf" align="middle" width="100%" height="25%"></object>
    <p></p>
    <embed  id="id-embed"  src="/JavaScriptAPI/MyFiller.pdf"  type="application/pdf"
         align="middle" width="100%" height="25%"/>
    <p></p>
  </body>
</html>
```

JavaScript code to test the WebQueue API.

WebQueue Code Sample

```
var m_queue = null ;

function queueOpen ()
{
    m_queue = LUICONNECTOR.createWebQueue () ;
    var host = { endpoint: "ws://chicago:7563/service/webqueue.jsp", profile: "*PRD" } ;
    var service = {queue: "REPORTS",
                   parameters: [{name: 'USERPARAM1', value: "XXX" } ] } ;
    /*
        The open function is not blocking and the code continues.
        The open is done asynchronously, you have no guarantee of an
        open connection when the call to the open function returns.
        You will be notified of an open connection via the onMessage call-back function.
        Optionally, the returned JSON object from the function call can only be used
        to confirm whether the local API call was successful.
    */

    var response = m_queue.open ( host, service, onMessage ) ;
    console.log ( JSON.stringify ( response ) ) ;
}

function queueClose ()
{
    if ( m_queue === null )
    {
        return ;
    }
    /*
        The close function is not blocking and the code continues.
        The close is done asynchronously, you have no guarantee of
        a closed connection when the call to the close function returns.
        You will be notified of a closed connection via the onMessage
        call-back function.
        Optionally, the returned JSON object from the function call
        can only be used to confirm whether the local API call was successful.
    */
    // var response = m_queue.close () ;
    m_queue.close ( 4500, "MY REASON" ) ;
    m_queue = null ;
```

WebQueue Code Sample

```
    // console.log ( JSON.stringify ( response ) ) ;
}

function queueSend ()
{
    if ( m_queue === null )
    {
        return ;
    }
    /*
        The send function is not blocking and the code continues.
        The send is done asynchronously, you have no guarantee a
        message has been sent when the call to the send function returns.
        You will be notified of an accepted or rejected sent message via the
        onMessage call-back function.
        Optionally, the returned JSON object from the function call
        can only be used to confirm whether the local API call was successful.
    */
    var message = 3456 ;
    var message = true ;
    var message = {text: "ABC"} ;
    // var message = "\u65E5\u672C\u8A9E" ;
    message = "A" ;
    for ( var i=0; i < 256 * 1024; i++ )
    {
        message = message + "B" ;
    }
    message = message + "C" ;
    // var response = m_queue.send ( 23, message ) ;
    // console.log ( JSON.stringify ( response ) ) ;
    /*
        Note: these message will be sent in the order of their execution
    */
    m_queue.send ( 23, "Test message" ) ;
    m_queue.send ( "ZXY", "Test message" ) ;
    m_queue.send ( { id:45}, "Test message" ) ;
    m_queue.send ( "LONG", message ) ;
    var file = document.getElementById ( 'id-file' ).files[0] ;
    if ( file instanceof File )
    {
```

WebQueue Code Sample

```
        /*
            Note: the send file is asynchronous due to the nature of
                  the FileReader object, so the send file message will be out of
                  execution sequence to other send calls
        */
        m_queue.send ( "DE47",
                  {type: "action", action: "sendfile", name: file.name,
                   mimetype: "application/octet-stream", value: "Some customer data"},
            file ) ;
    }
    /*
        Note: you cannot send any more messages until the asynchronous file send
              has completed, you will be changing the internal state of
              the m_object
    */
    console.log ( "IS LOCKED " + m_queue.isLocked () ) ;
    m_queue.send ( "AFTERFILE", "AAA" ) ;
}
function onMessage ( message, binaryData )
{
    /*
        - Host errors
        - Host messages
        - Local API errors
    */
    switch ( message.type )
    {
        case "error":
        {
            onMessageError ( message ) ;
            return ;
        }
        case "status":
        {
            switch ( message.status )
            {
                case "OPEN":
                {
                    onMessageOpen ( message ) ;
                    return ;
```

WebQueue Code Sample

```
                }
                case "CLOSE":
                {
                    onMessageClose ( message ) ;
                    return ;
                }
                case "READY":
                {
                    onMessageReady ( message ) ;
                    return ;
                }
                case "PUSH":
                {
                    onMessagePush ( message, binaryData ) ;
                    return ;
                }
                case "ACCEPT":
                {
                    onMessageAccept ( message ) ;
                    return ;
                }
                case "REJECT":
                {
                    onMessageReject ( message ) ;
                    return ;
                }
                default:
                {
                    console.log (
                     "onMessage 'status DEFAULT' " + JSON.stringify ( message ) ) ;
                    return ;
                }
            }
        }
        default:
        {
            /*
                switch ( property ) is null, undefined or not evaluated
            */
            console.log ( "onMessage 'type DEFAULT' " + JSON.stringify ( message ) ) ;
```

WebQueue Code Sample

```
            return ;
        }
    }
}

function onMessageError ( message )
{
    /*
        Host or Local API error
    */
    var reason = message.message ;
    console.log ( "onMessage 'type ERROR' " + JSON.stringify ( message ) ) ;
}

function onMessageOpen ( message )
{
    /*
        WebSocket has established a socket connection to WebQueue host
    */
    console.log ( "onMessage 'status OPEN' " + JSON.stringify ( message ) ) ;
}

function onMessageReady ( message )
{
    /*
        WebQueue host has completed the initialization process and
        is ready to send and receive messages
    */
    var qid = message.qid ;
    var queue = message.queue ;
    console.log ( "onMessage 'status READY' " + JSON.stringify ( message ) ) ;
}

function onMessageClose ( message )
{
    /*
        Connection has closed, either client or host initiated
    */
    var code = message.code ;
    var clean = message.clean ;
```

Content © LANSA Group
All Rights Reserved

WebQueue Code Sample

```
    var reason = message.reason ;
    console.log ( "onMessage 'status CLOSE' " + JSON.stringify ( message ) ) ;
}

function onMessageReject ( message )
{
    /*
        Message sent to host has been rejected
    */
    var code = message.code ;
    var reason = message.reason ;
    var messageId = message.correlationid ;
    console.log ( "onMessage 'status REJECT' " + JSON.stringify ( message ) ) ;
}

function onMessageAccept ( message )
{
    /*
        Message sent to host has been accepted
    */
    var messageId = message.correlationid ;
    console.log ( "onMessage 'status ACCEPT' " + JSON.stringify ( message ) ) ;
}

function onMessagePush ( message, binaryData )
{
    /*
        Host has pushed a message with optional binary data
    */
    // String or JSON object
    var value = message.value ;
    var sender = message.sender ;
    var messageId = message.messageid ;
    var qid = messageId.qid ;
    var cid = messageId.cid ;
    // Epoch time in milliseconds, IBM i has a time resolution of 1 ms
    var tid = messageId.tid ;
    /*
       The Number.MAX_SAFE_INTEGER constant represents the maximum safe integer
       in JavaScript (2^53 - 1) 9007199254740991
```

WebQueue Code Sample

```
    */
    var date = new Date( tid ) ;
    /*
        tid : 1434685216396
        date: Fri Jun 19 2015 13:40:16 GMT+1000 (AUS Eastern Standard Time)
    */
    // console.log ( date.toDateString() + " " + date.toTimeString() ) ;
    console.log ( "onMessage 'status PUSH' " + cid + " " + JSON.stringify ( message ) ) ;
    if ( binaryData )
    {
    /*
        Note: binaryData object is an instanceof Uint8Array object message object is
              {"sversion": "1.2.1", "value":{"name": "abc.pdf",
               "mimetype": application/pdf", "action": sendfile",
               "type": "action"}, "status": "OK", "sender": "", "type": "status"}
        ====================================================================
        You need to pay attention to Blob and URL creation and releasing
        Use 'chrome://blob-internals' to view the current Blob allocations
        Under chrome, blobs take a long time to be garbage collected
        ====================================================================
    */

    /*
        Unicode Text
    */
        if ( message.value.mimetype === "text/plain" )
        {
            if ( true )
            {
            /*
                Plain Text Test
            */
                var text = LUIENCRYPT.decodeUTF8Uint8Array ( binaryData, true ) ;
            }
            return ;
        }
        /*
            Image
        */
        if ( message.value.mimetype === "image/jpeg" )
```

WebQueue Code Sample

```
        {
            if ( true )
            {
                /*
                    Image Tag Test
                */
                var objectURL = window.URL.createObjectURL ( new Blob( [ binaryData ],
                                { type:message.value.mimetype } ) ) ;
                try
                {
                    var img = document.getElementById ( 'id-image' ) ;
                    img.onload = function ()
                    {
                        /*
                            Revoke objectURL
                        */
                        // console.log ( "image onload" ) ;
                        window.URL.revokeObjectURL ( this.src ) ;
                    }
                    img.src = window.URL.createObjectURL ( new Blob( [ binaryData ],
                                { type:message.value.mimetype } ) ) ;
                }
                catch ( e )
                {
                    window.URL.revokeObjectURL ( objectURL ) ;
                    console.log ( "FAILED <img> " + e ) ;
                    return ;
                }
            }
            return ;
        }
        /*
            PDF
        */
        if ( message.value.mimetype === "application/pdf" )
        {
            if ( false )
            {
                /*
                    Anchor Test
```

Content © LANSA Group
All Rights Reserved

WebQueue Code Sample

```
            */
            var objectURL = window.URL.createObjectURL ( new Blob( [ binaryData ],
                            { type:message.value.mimetype } ) ) ;
            try
            {
                var anchor = document.getElementById ( "id-save" ) ;
                anchor.download = message.value.name ;
                anchor.innerHTML = "Click to Save" ;
                if ( anchor.href )
                {
                    /*
                        Revoke previous object URL
                    */
                    window.URL.revokeObjectURL ( anchor.href ) ;
                }
                anchor.href = objectURL ;
            }
            catch ( e )
            {
                window.URL.revokeObjectURL ( objectURL ) ;
                console.log ( "FAILED <a> " + e ) ;
            }
        }
        if ( true )
        {
            /*
                Iframe Test
                Works: Chrome and Firefox
                Fails: IE
                Notes: IE does not throw any exception
            */
            var objectURL = window.URL.createObjectURL ( new Blob( [ binaryData ],
                            { type:message.value.mimetype } ) ) ;
            try
            {
                var iframe = document.getElementById ( 'id-frame' ) ;
                iframe.onload = function ()
                {
                    /*
                        Revoke objectURL
```

WebQueue Code Sample

```
                    */
                    // console.log ( "iframe onload" ) ;
                    window.URL.revokeObjectURL ( this.src ) ;
                }
                iframe.src = objectURL ;
            }
            catch ( e )
            {
                window.URL.revokeObjectURL ( objectURL ) ;
                console.log ( "FAILED <iframe> " + e ) ;
            }
        }
        if ( true )
        {
            /*
                Object Test
                Works: Chrome and Firefox
                Fails: IE
                Notes: IE throws an 'Access denied' exception
            */
            var objectURL = window.URL.createObjectURL ( new Blob( [ binaryData ],
                                { type:message.value.mimetype } ) ) ;
            try
            {
                var control = document.getElementById ( 'id-object' ) ;
                control.data = objectURL ;
            }
            catch ( e )
            {
                /*
                    IE 'Access denied' exception
                */
                window.URL.revokeObjectURL ( objectURL ) ;
                console.log ( "FAILED <object> " + e ) ;
            }
        }
        if ( true )
        {
                /*
                    Embed Test
```

WebQueue Code Sample

```
                        Works: none
                        Fails: IE, Chrome and Firefox
                        Notes: IE does not throw any exception
                    */
                  var objectURL = window.URL.createObjectURL ( new Blob( [ binaryData ],
                                  { type:message.value.mimetype } ) ) ;
                    try
                    {
                        var control = document.getElementById ( 'id-embed' ) ;
                        control.src = objectURL ;
                    }
                    catch ( e )
                    {
                        window.URL.revokeObjectURL ( objectURL ) ;
                        console.log ( "FAILED <embed> " + e ) ;
                    }
                }
                if ( false )
                {
                    /*
                        Open Window Test
                        Works: Chrome and Firefox
                        Fails: IE
                        Notes: IE throws an 'Access denied' exception
                               Safari does not allow window.open
                               unless it is from a click event
                    */
                  var objectURL = window.URL.createObjectURL ( new Blob( [ binaryData ],
                                  { type:message.value.mimetype } ) ) ;
                    try
                    {
                        var win = window.open ( objectURL ) ;
                        if ( win == null )
                        {
                            /*
                                Window not opened
                            */
                            window.URL.revokeObjectURL ( objectURL ) ;
                            // console.log ( "window open test null window" ) ;
                            return ;
```

WebQueue Code Sample

```
                    }
                    /*
                        https://developer.mozilla.org/en-US/docs/Web/API/Window/open
                        Note that remote URLs won't load immediately
                        When window.open() returns, the window always contains
                        about:blank
                        The actual fetching of the URL is deferred and starts
                        after the current script block finishes executing
                        The window creation and the loading of the referenced
                        resource are done asynchronously
                        Window has been opened, release object URL
                    */
                    window.URL.revokeObjectURL ( objectURL ) ;
                    // setTimeout ( function () { window.URL.revokeObjectURL
                       ( objectURL ) }, 2000 ) ;
                    return ;
                }
                catch ( e )
                {
                    /*
                        IE 'Access denied' exception
                    */
                    window.URL.revokeObjectURL ( objectURL ) ;
                    console.log ( "FAILED window.open " + e ) ;
                }
            }
            return ;
        }
        return ;
    }
}
```

Configuration required to test the WebQueue API.

WebQueue Configuration

```
<match uri="/service/webqueue.jsp"
    class="com.lansa.mobile.service.HTTPServiceWebQueue"
    <host name="LOCAL" system="LOCALHOST">
        <parameter name="service.queue.REPORTS"
         value="{
```

WebQueue Configuration

```
            limit:10485760,
            read: {object:'TSTWEB/RPT{QUEUEID}R',length:1024},
            write:{object:'TSTWEB/RPT{QUEUEID}W',length:1024},
            upload:{folder:'/upload/reports/{QUEUEID}'},
            connect:{
                     program:'TSTLIB/TSTCONNECT',
                     parameters:[{name:'QUEUE',      length:32},
                                 {name:'QUEUEID',    length:6},
                                 {name:'QUEUEUSER',  length:10},
                                 {name:'QUEUERLIB',  length:10},
                                 {name:'QUEUERNME',  length:10},
                                 {name:'QUEUEWLIB',  length:10},
                                 {name:'QUEUEWNME',  length:10},
                                 {name:'USERPARAM1', length:20}]
                     }
         }"/>
    </host>
</match>
```

Code for the TSTCONNECT program (IBM i Command Language code).

WebQueue TSTCONNECT Source Code

```
PGM  PARM(&QUEUE &QUEUEID &QUEUEUSER &QUEUERLIB &QUEUERNME  +
          &QUEUEWLIB &QUEUEWNME &USERPARAM1)

DCL VAR(&QUEUE)      TYPE(*CHAR) LEN(32)
DCL VAR(&QUEUEID)    TYPE(*CHAR) LEN(6)
DCL VAR(&QUEUEUSER)  TYPE(*CHAR) LEN(10)
DCL VAR(&QUEUERLIB)  TYPE(*CHAR) LEN(10)
DCL VAR(&QUEUERNME)  TYPE(*CHAR) LEN(10)
DCL VAR(&QUEUEWLIB)  TYPE(*CHAR) LEN(10)
DCL VAR(&QUEUEWNME)  TYPE(*CHAR) LEN(10)
DCL VAR(&USERPARAM1) TYPE(*CHAR) LEN(20)
DCL VAR(&DTA)        TYPE(*CHAR) LEN(1024)
DCL VAR(&LEN)        TYPE(*DEC)  LEN(5 0) VALUE(1024)

IF (&QUEUE *EQ 'REPORTS' ) THEN(DO)
    SBMJOB     CMD(CALL PGM(TSTLIB/TSTREPORTS) PARM(&QUEUE +
                   &QUEUEID &QUEUEUSER &QUEUERLIB &QUEUERNME &QUEUEWLIB +
                   &QUEUEWNME &USERPARAM1)) JOB(REPORTJOB)
```

Content © LANSA Group
All Rights Reserved

WebQueue TSTCONNECT Source Code

```
   GOTO END
ENDDO

CHGVAR VAR(&DTA) VALUE('{type: "action",action: "close",code:4100,+
reason: "No connect program associated with queue"}')
CALL PGM(QSNDDTAQ) PARM(&QUEUERNME &QUEUERLIB &LEN &DTA)
MONMSG  MSGID(CPF9801) /* DOES NOT EXIST */

END: ENDPGM
```

Code for the TSTREPORTS program (IBM i Command Language code).

WebQueue TSTREPORTS Source Code

```
PGM  PARM(&QUEUE &QUEUEID &QUEUEUSER &QUEUERLIB &QUEUERNME  +
          &QUEUEWLIB &QUEUEWNME &USERPARAM1)

DCL VAR(&QUEUE)      TYPE(*CHAR) LEN(32)
DCL VAR(&QUEUEID)    TYPE(*CHAR) LEN(6)
DCL VAR(&QUEUEUSER)  TYPE(*CHAR) LEN(10)
DCL VAR(&QUEUERLIB)  TYPE(*CHAR) LEN(10)
DCL VAR(&QUEUERNME)  TYPE(*CHAR) LEN(10)
DCL VAR(&QUEUEWLIB)  TYPE(*CHAR) LEN(10)
DCL VAR(&QUEUEWNME)  TYPE(*CHAR) LEN(10)
DCL VAR(&USERPARAM1) TYPE(*CHAR) LEN(20)
DCL VAR(&DTA)        TYPE(*CHAR) LEN(1024)
DCL VAR(&LEN)        TYPE(*DEC)  LEN(5 0) VALUE(1024)

IF (&QUEUE *EQ 'REPORTS' ) THEN(DO)

ENDDO

CHGVAR VAR(&DTA) VALUE('TSTREPORTS STAGE 1 ')
CALLSUBR  SUBR(SEND)

DLYJOB DLY(1)
CHGVAR VAR(&DTA) VALUE('TSTREPORTS STAGE 2 ')
CALLSUBR  SUBR(SEND)

GOTO END
```

WebQueue TSTREPORTS Source Code

```
DLYJOB DLY(1)
CHGVAR VAR(&DTA) VALUE('{type: "action",action: "close",code:4000,+
reason: "TSTREPORTS has completed"}')
CALLSUBR  SUBR(SEND)

SUBR SUBR(SEND)
  CALL PGM(QSNDDTAQ) PARM(&QUEUERNME &QUEUERLIB &LEN &DTA)
  MONMSG  MSGID(CPF9801) /* DOES NOT EXIST */
ENDSUBR

END: ENDPGM
```

Code for the TSTSENDFILE program (IBM i Command Language code).

WebQueue TSTSENDFILE Source Code

```
PGM PARM(&QUEUERNME)

DCL VAR(&QUEUERNME)  TYPE(*CHAR) LEN(10)
DCL VAR(&QUEUERLIB)  TYPE(*CHAR) LEN(10)
DCL VAR(&DTA)        TYPE(*CHAR)          LEN(1024)
DCL VAR(&LEN)        TYPE(*DEC)  LEN(5 0) VALUE(1024)
DCL VAR(&COUNT)      TYPE(*INT)  LEN(2)

CHGVAR VAR(&QUEUERLIB) VALUE('TSTWEB')

DOFOR  VAR(&COUNT) FROM(1) TO(1)
CHGVAR VAR(&DTA) +
VALUE('{type: "action", action: "sendfile",+
path: "/devjsm/instance/www/instance/htdocs/JavaScriptAPI/MyImage.jpg",+
name: "abc.jpg",mimetype: "image/jpeg"}')
CALLSUBR  SUBR(SEND)
ENDDO

CHGVAR VAR(&DTA) +
VALUE('{type: "action",action: "sendfile",+
path: "/devjsm/instance/www/instance/htdocs/JavaScriptAPI/MyPDF.pdf",+
name: "abc.pdf",mimetype: "application/pdf"}')
CALLSUBR  SUBR(SEND)

CHGVAR VAR(&DTA) +
```

WebQueue TSTSENDFILE Source Code

```
VALUE('{type: "action",action: "sendfile",+
path: "/devjsm/instance/www/instance/htdocs/JavaScriptAPI/MyText.txt",+
name: "abc.txt",mimetype: "text/plain"}')
CALLSUBR  SUBR(SEND)

GOTO END

SUBR SUBR(SEND)
  CALL PGM(QSNDDTAQ) PARM(&QUEUERNME &QUEUERLIB &LEN &DTA)
  MONMSG MSGID(CPF9801) EXEC(SNDPGMMSG MSG('Queue DOES NOT EXIST'))
ENDSUBR

END: ENDPGM
```

Viewing PDF documents in a browser

Developers can use the WebQueue API to download PDF documents and present them for viewing in a Web application.

HTML page for viewing PDF documents.

HTML page to test downloading and viewing PDF files

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>PDF ViewPort Example</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <script type="text/javascript" src="pdf.js"></script>
    <script type="text/javascript" src="lui-pdf.js"></script>
    <script type="text/javascript">

    </script>
  </head>
  <body>
    <button onclick="PDFOpen()">Open</button>
    <button onclick="PDFClose()">Close</button>
    <button onclick="PDFPrevious()">Previous</button>
    <button onclick="PDFNext()">Next</button>
```

HTML page to test downloading and viewing PDF files

```
    <button onclick="PDFZoom( +0.25 )">+</button>
    <button onclick="PDFZoom( -0.25 )">-</button>
    <button onclick="PDFMetaData()">MetaData</button>
    <p></p>
    <div id= "id-page" style= "border:1px solid black; border-width:2; width:400px;
                              height:400px; overflow:auto;">
        <canvas id="id-canvas" width="600" height="800"></canvas>
    </div>
  </body>
</html>
```

JavaScript for viewing PDF documents.

JavaScript to test downloading and viewing PDF files

```
PDFJS.disableRange = true ;
PDFJS.disableStream = true ;
PDFJS.disableAutoFetch = true ;
var m_viewPort = LUIPDF.createViewPort () ;

function PDFOpen ()
{
    m_viewPort.open ( document.getElementById ( 'id-canvas' ),
                      'MyPDF.pdf', illustrator ) ;
}

function PDFClose ()
{
    m_viewPort.close ( true ) ;
}

function PDFPrevious ()
{
    m_viewPort.previous () ;
}

function PDFNext ()
{
    m_viewPort.next () ;
}
```

JavaScript to test downloading and viewing PDF files

```
function illustrator ( pageNumber, page, canvas, context, viewport )
{
    /*
        Canvas
            0,0 ------ X ---→
            |
            |
            Y
            |
            |
        Use PDF point locations, so the drawing stays at the same location when zooming.
    */
    console.log ( "Illustrate page " + pageNumber ) ;
    // console.log ( "width: " + viewport.width + " height: " + viewport.height ) ;
    // var pointA = viewport.convertToViewportPoint ( 0, 0 ) ;   // x1,y1
    var pointA = viewport.convertToViewportPoint ( 200, 600 ) ; // x1,y1
    var pointB = viewport.convertToViewportPoint ( 200, 400 ) ; // x2,y2
    context.beginPath() ;
        context.lineWidth = 4 ;
        context.moveTo( pointA[0], pointA[1] ) ; // x1, y1
        context.lineTo( pointB[0], pointB[1] ) ; // x2, y2
        context.strokeStyle = "orange";
        context.stroke() ;
    context.closePath() ;
}

function PDFZoom ( delta )
{
    m_viewPort.zoom ( delta ) ;
}

function PDFMetaData ()
{
    var pdfDocument = m_viewPort.getDocument () ;
    if ( pdfDocument == null )
    {
        return ;
    }
    pdfDocument.getMetadata().then( function ( data )
    {
```

Content © LANSA Group
All Rights Reserved

JavaScript to test downloading and viewing PDF files

```
        var info = {
                      'title':           data.info.Title,
                      'author':          data.info.Author,
                      'subject':         data.info.Subject,
                      'keywords':        data.info.Keywords,
                      'creationDate':    data.info.CreationDate,
                      'modificationDate': data.info.ModDate,
                      'creator':         data.info.Creator,
                      'producer':        data.info.Producer,
                      'version':         data.info.PDFFormatVersion
                   } ;
        var metadata = data.metadata ;
        console.log ( JSON.stringify ( info ) ) ;
        // console.log ( JSON.stringify ( metadata ) ) ;
    } ) ;
}
```

Content © LANSA Group
All Rights Reserved

Appendices

Glossary

Table 11 (page 52) presents definitions for abbreviations and terms used in this guide.

Table 11: Abbreviations and Terms

Abbreviations and Terms	Definitions and Explanations
CJS	ConnectorJS
CJS Server	ConnectorJS Server
Connector configuration	A connector configuration is a set of directives in the configuration that describe an API call. Connector configurations contain one or more parameter definitions.
Connector element	A connector element comprises connector tags that define connector APIs and wrap the parameters associated with a connector. <connector name= "name" Parameters associated with the connector </connector>
Connector name	Connector configurations have unique names. The WebQueue connector name is webqueue ConnectorJS uses the name to associate an API call with its configuration.
ConnectorJS Server	This collective term refers to the ConnectorJS components that reside on a server.
CORS	Cross-origin request sharing
Data queue	Program data queues are an implementation of message queues on IBM i servers. CJS uses program data queues to manage the message exchanges between web applications and CJS Server.
Directives	Configuration directives are the parameters and settings that control the behaviour of the ConnectorJS APIs.
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure

Abbreviations and Terms	Definitions and Explanations
IFS	The Integrated File System is a part of the IBM i operating system. It supports stream input/output and storage management capabilities similar to personal computer and UNIX operating systems.
JMS API	Java Message Service API
JSM	Java Services Manager (ConnectorJS Server administration services)
JSON	JavaScript Object Notation
Parameter definition	A parameter definition is a set of directives in the configuration that describe keyword and values ConnectorJS uses to respond to an API call. Connector configurations may include one or more parameter definitions.
Response	Response refers to the data returned to a Web application by ConnectorJS. The structure of the content of the response is a JSON object.
System directory	System directory is a synonym for the JSM system folder.
System folder	The JSM system folder location is: LUI Connector/jsm/instance/system
TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security
WebSocket	The WebSocket protocol provides full-duplex communications over a TCP connection.
XHR	XMLHttpRequest

Assumed and prerequisite knowledge

This guide assumes knowledge of the following topics.

Table 12: Assumed and Prerequisite Knowledge

Subject Matter	Explanations
IBM i operating system	Developers require an understanding of IBM i servers, libraries, programs, commands, object authority, program data queues and the Integrated File System (IFS).
JavaScript	Developers require a knowledge of coding JavaScript and building Web applications.

Subject Matter	Explanations
Messaging software	This guide assumes developers understand how to use program data queues and/or install, configure and operate messaging software such as IBM WebSphere MQ.

Program data queue size

When designing messaging applications, developers should consider program data queue size limits to determine maximum entry length and queue size. Queue size is a storage allocation, e.g. *MAX2GB allocates 2 GB of storage for the queue. Maximum entry length and queue size define the maximum number of entries, i.e. queue size divided by maximum entry length determines the maximum number of entries that fit into the storage space allocated to the queue. The maximum number of entries for a 64-byte message using a 40-byte key and no sender id is 12,201,433.

CRTDTAQ DTAQ(MYLIB/TSTRCVQ) MAXLEN(64) SEQ(*KEYED) KEYLEN(40) SENDERID(*NO) SIZE(12201433) AUTORCL(*YES)

Reading entries from a queue releases storage for new entries.

Factors that contribute to the number of entries in a queue are the rate of arrival of new entries and the rate of reading entries from the queue. Queue size must balance these factors to achieve optimum performance.

Content © LANSA Group
All Rights Reserved