Checklist: How to Build C Clients Using Kaazing WebSocket Gateway

This checklist provides the steps necessary to build AMQP clients to communicate with Kaazing WebSocket Gateway:

#	Step	Topic or Reference
1	Learn about the Kaazing WebSocket Gateway C AMQP client.	Overview of the Kaazing WebSocket Gateway AMQP C Client Library
2	Learn how to use the Gateway C AMQP Client Library and the supported APIs.	Use the Kaazing WebSocket Gateway C AMQP Client Library
3	Learn how to establish trust between your C client and the Gateway by using client-side TLS/SSL digital certificates.	Require Clients to Provide Certificates to the Gateway

Overview of AMQP 0-9-1

Advanced Message Queuing Protocol (AMQP) is an open standard for messaging middleware that was originally designed by the financial services industry to provide an interoperable protocol for managing the flow of enterprise messages. To guarantee messaging interoperability, AMQP 0-9-1 defines both a wire-level protocol and a model—the AMQP Model—of messaging capabilities.

The AMQP Model defines three main components:

1. *Exchange*: clients publish messages to an exchange
2. *Queue*: clients read messages from a queue
3. *Binding*: a mapping from an exchange to a queue

An AMQP client connects to an AMQP broker and opens a *channel*. Once the channel is established, the client can send messages to an exchange and receive messages from a queue. To learn more about AMQP functionality, take a look at the Real-Time Interactive Guide to AMQP, an interactive guide that takes you step-by-step through the main features of AMQP version 0-9-1.

For more information about AMQP, visit http://www.amqp.org.

WebSocket and AMQP

WebSocket enables direct communication from the Web client to an AMQP broker via the Gateway. The Gateway radically simplifies Web application design by providing the AMQP client libraries for C client technologies.

Using the AMQP client libraries, you can take advantage of the AMQP features, making the C Web client a first-class citizen in AMQP systems.

Overview of the Kaazing WebSocket Gateway AMQP Client Library

Kaazing WebSocket Gateway - AMQP Edition includes AMQP client libraries that allow clients to subscribe and publish messages to a message broker using AMQP. With the Kaazing WebSocket Gateway AMQP client libraries, you can leverage WebSocket in your application by building a C client. This WebSocket client then enables communication between your application and the message broker, as shown in the following figure:

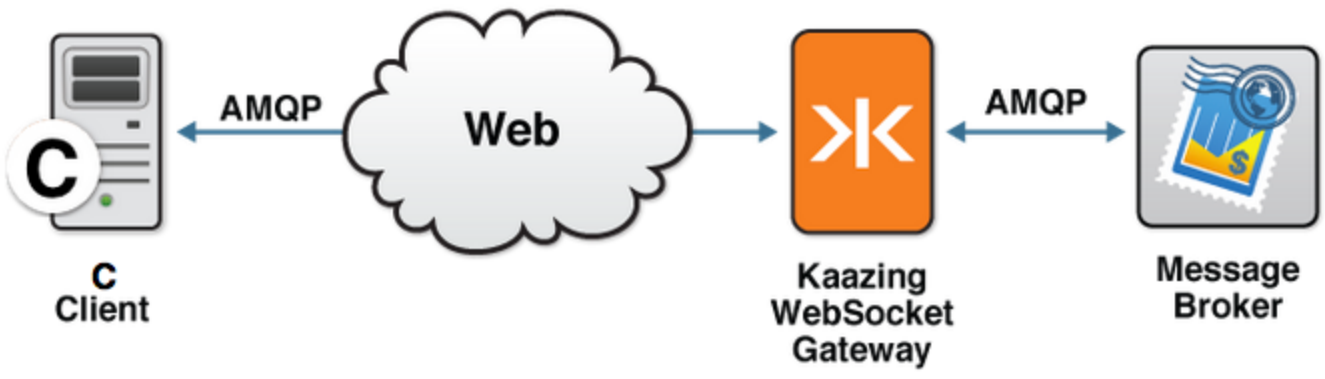


Figure: Kaazing WebSocket Gateway C AMQP Client Connection

Starting an AMQP Broker

There are a wide variety of AMQP brokers available that implement different AMQP versions. For example, RabbitMQ, Apache Qpid, OpenAMQ, Red Hat Enterprise MRG, UMQ, and Zyre. If you do not have an AMQP broker installed yet, you can use the Apache Qpid AMQP broker included in the Gateway download package, that supports AMQP version 0-9-1.

To set up and start the Apache Qpid broker on your system, perform the steps described in Setting Up Kaazing WebSocket Gateway.

Note: The AMQP client libraries are compatible with AMQP version 0-9-1. Refer your AMQP broker documentation for information about certified AMQP versions.

Taking a Look at the AMQP Demo

Once the AMQP broker is set up, take a look at a demonstration that was built with the C version of the AMQP client library. To see this demo in action, perform the following steps:

1. Start the Kaazing WebSocket Gateway and Apache Qpid as described in Setting Up Kaazing WebSocket Gateway.
2. In a browser, navigate to http://localhost:8001/demo/amqp/c/?d=amqp-c and follow the instructions. From this page, you can navigate to additional AMQP demos for different client technologies (Java, Silverlight, .NET Framework, and Adobe Flash and Flex).

Use the Kaazing WebSocket Gateway C AMQP Client Library

In this procedure, you will learn how to use the Kaazing WebSocket Gateway C AMQP Client Library and the supported APIs:

- Build the client using the Eclipse IDE as described in To Use the Kaazing WebSocket Gateway C AMQP Client Library.
- See examples for producing and consuming messages, including OpenSSL: Examples.

This topic shows you how to set up your development environment and perform the following coding steps:

1. Set up your development environment.
2. Review the common C AMQP programming steps.
3. Import the libraries.
4. Create the WebSocket connection to the Gateway.
5. Connect and log into an AMQP broker.
6. Create channels.
7. Declare an exchange.
8. Declare a queue.
9. Bind an exchange to a queue.
10. Consume messages.
11. Publish messages.
12. Handle Exceptions

Before You Begin

This procedure is part of Checklist: How to Build C Clients Using Kaazing WebSocket Gateway:

1. Overview of the Kaazing WebSocket Gateway AMQP Client Library
2. **Use the Kaazing WebSocket Gateway C AMQP Client Library**

3. Require Clients to Provide Certificates to the Gateway

Components and Tools

Before you get started, review the components and tools used to build your AMQP C client, described in the following table:

Component or Tool	Description	Location
Kaazing WebSocket Gateway - AMQP Edition	For a detailed description of the Kaazing WebSocket Gateway - AMQP Edition, see http://kaazing.com/products/editions/kaazing-websocket-gateway-amqp.	http://developer.kaazing.com/downloads/amqp-edition-download/
AMQP broker	The full Kaazing WebSocket Gateway - AMQP Edition download (Gateway + Documentation + Demos) includes the Apache Qpid broker.	Qpid_HOME: The installer creates the default Qpid_HOME destination in C:\Program Files\Kaazing\amqp*version*\qpid (Windows) or /usr/share/kaazing/amqp/*version*/(Linux). The full Kaazing WebSocket Gateway - AMQP Edition download contains the Apache Qpid broker folder in its root directory.
Kaazing WebSocket Gateway AMQP C Client library files	The C header files (.h) and the shared object files (.so) that make up the Kaazing WebSocket Gateway AMQP C Client library.	AMQP header files (.h) and library (.so): *GATEWAY_HOME*/lib/client/c/amqp/*OS*/include *GATEWAY_HOME*/lib/client/c/amqp/*OS*/lib WebSocket header files (.h) and library (.so): *GATEWAY_HOME*/lib/client/c/websocket/*OS*/include *GATEWAY_HOME*/lib/client/c/websocket/*OS*/lib *OS* represents the files for your operating system. For some operating systems, there might be subfolders, such as: *GATEWAY_HOME*/lib/client/c/amqp/Ubuntu_precise/lib/x86_64-linux-gnu
CDT (C/C++ Development Tooling) plugin for Eclipse or Eclipse IDE for C/C++ Developers	The Eclipse IDE is used to develop, compile, and test the AMQP C client. You have two options when configuring Eclipse: ● If you have an installed Eclipse package (for example, Eclipse for Java Developers), you can install the CDT plug-in as described here: http://www.eclipse.org/cdt/downloads.php. ● If you do not have an Eclipse package, you can download Eclipse IDE for C/C++ Developers from http://www.eclipse.org/downloads, and install the downloaded files.	Eclipse IDE for C/C++ Developers: http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/keplersr2 CDT (C/C++ Development Tooling) plugin: http://www.eclipse.org/cdt/downloads.php
OpenSSL	OpenSSL is required by the C client to connect with the Gateway securely over TLS/SSL. See the OpenSSL examples: amqp_sendstring_websocket_ssl.c, and amqp_sendstring_websocket_ssl_callback.c.	OpenSSL is typically installed on most operating systems; however, if you are using a Linux distribution, you might need to install it. In most cases, you can simply update and upgrade your Linux distribution (`sudo apt-get update && sudo apt-get upgrade`), or install the OpenSSL development package (`sudo apt-get install openssl libssl-dev`). For more information, see http://www.openssl.org/support/faq.html and http://www.openssl.org/source/.

To Use the Kaazing WebSocket Gateway C AMQP Client Library

Use the following steps to build and run an AMQP C client using the Kaazing WebSocket Gateway AMQP C library.

1. Setting Up Your Development Environment.

   To develop applications using the Kaazing WebSocket Gateway C AMQP client libraries, you must configure the Gateway to communicate with an AMQP broker.

   Note: If you have the Kaazing WebSocket Gateway running on localhost and if you have an AMQP broker running on localhost at the default AMQP port 5672, you do not have to configure anything to see the AMQP demos and the interactive AMQP guide.

   The following is an example of the default configuration element for the AMQP service in the Kaazing WebSocket Gateway bundle, as specified i the configuration file *GATEWAY_HOME*/conf/gateway-config.xml:

```
<!-- Proxy service to AMQP server -->
<service>
  <accept>ws://localhost:8001/amqp</accept>
  <connect>tcp://localhost:5672</connect>
   <type>amqp.proxy</type>
  <cross-site-constraint>
    <allow-origin>http://localhost:8001</allow-origin>
  </cross-site-constraint>
</service>
```

   In this case, the service is configured to accept WebSocket AMQP requests from the browser at ws://localhost:8001/amqp and proxy those requests to a locally installed AMQP broker (localhost) at port 5672.

   To configure the Gateway to accept WebSocket requests at another URL or to connect to a different AMQP broker, you can edit *GATEWAY_HOME*/conf/gateway-config.xml, update the values for the accept elements, change the connect property, and restart the Gateway. For example, the following configuration configures the Gateway to accept WebSocket AMQP requests at ws://www.example.com:80/amqp and proxy those requests to an AMQP broker (amqp.example.com) on port 5672.

```
<!-- Proxy service to AMQP server -->
<service>
    <accept>ws://www.example.com:80/amqp</accept>
    <connect>tcp://amqp.example.com:5672</connect>
    <type>amqp.proxy</type>
</service>
```

2. Review the common C AMQP programming steps.

   You can either build a single application that both publishes and consumes messages, or create two different applications to handle each action.

   The demo located at http://localhost:8001/demo/amqp/c/?d=amqp-c shows a single application that handles both actions. You can view the source code for this demo in *GATEWAY_HOME*/demo/c. Refer to the AMQP Client C API documentation for the complete list of all the AMQP command and callback functions.

   The common C AMQP programming steps are:

   a. Download, install and configure the Eclipse IDE.
   b. Create an Eclipse C project.
   c. Import AMQP C client libraries.
   d. Create source file for Gateway and AMQP broker connections and consuming messages.
   e. Create source file for publishing.

3. Download and install the Eclipse IDE for C/C++ Developers. This topic uses the Kepler package of Eclipse.

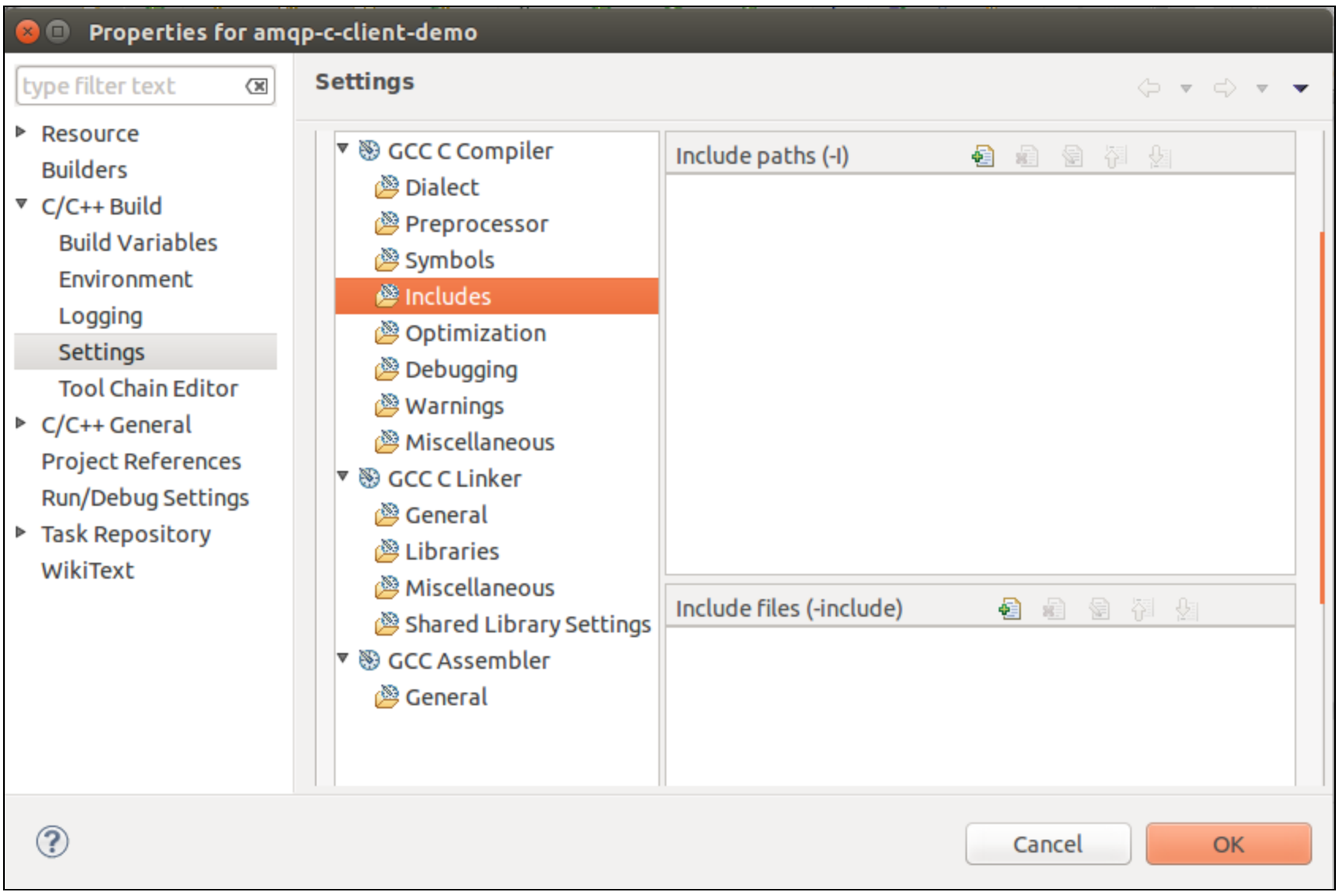
   Download the Eclipse IDE for C/C++ Developers from here: https://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/keplersr2

   If you have an installed Eclipse package (for example, Eclipse for Java Developers), you can install the CDT plug-in as described here: http://www.eclipse.org/cdt/downloads.php

4. Create an Eclipse C project.

a. Launch Eclipse and select a workspace.
b. In the welcome screen, click the Workbench icon.
c. Right-click in the empty space in **Project Explorer**, click **New**, and then click **C Project**. The **C Project** dialog appears.
d. In **Project name**, enter **amqp-c-client-demo**.
e. In **Project type**, expand the **Executable** folder and select **Empty Project**.
f. In **Toolchains**, select the compiler for your OS platform.
g. Click **Finish**. The new project appears.

5. Add AMQP C header files.
   a. Right-click the **amqp-c-client-demo** project in **Project Explorer** and click **Properties**.
   b. In the **Properties** window, click **Settings** under the **C/C++ Build** heading.
   c. In the **Settings** area, click **Tool Settings**.
   d. Click **Includes** under the compiler heading, for example **GCC C Compiler** (this heading might be different depending on the compiler you selected when creating the project).



   e. In **Include paths**, click the **Add** icon. The **Add directory path** dialog appears.
   f. Click **File System**, and then navigate to the Kaazing WebSocket Gateway C header folders and click **Open**:

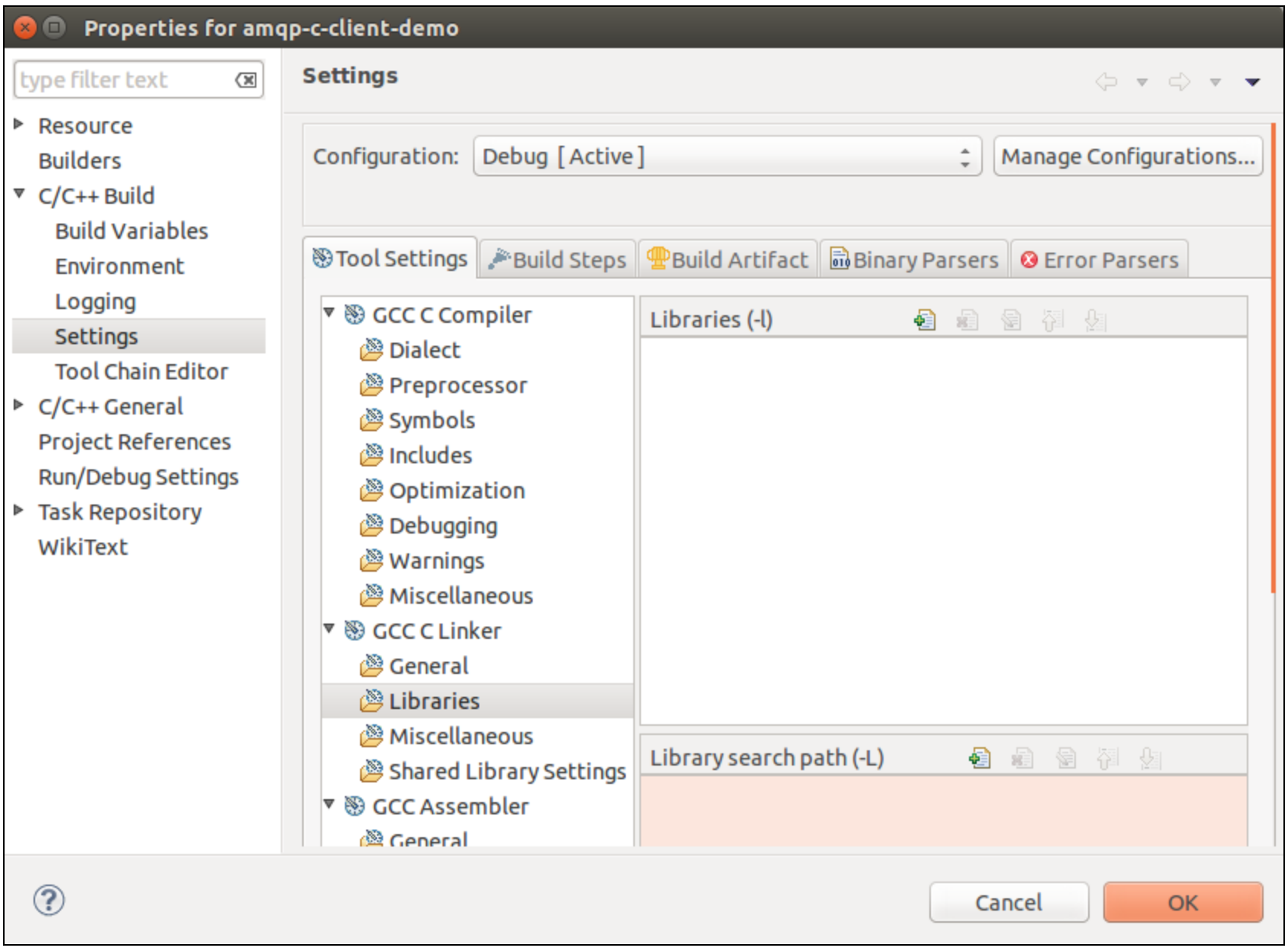
      GATEWAY_HOME/lib/client/c/amqp/*OS*/include

      GATEWAY_HOME/lib/client/c/websocket/*OS*/include

   g. Click **Apply**. The libraries appear in the project.

6. Add AMQP C shared object files.

a. In the **Properties** window for the project, click **Libraries** under the linker heading, for example, **GCC C Linker**.



b. In **Libraries**, click the **Add** icon. The **Enter Value** dialog appears.

c. Enter the name of the library, **qpid**, and click **OK**.

d. In **Library search path**, click the **Add** icon. The **add directory path** dialog appears.

e. Click **File system**, navigate to the location of each of the library folders and click **OK**:

GATEWAY_HOME/lib/client/c/amqp/*OS*/lib

GATEWAY_HOME/lib/client/c/websocket/Ubuntu_precise/lib

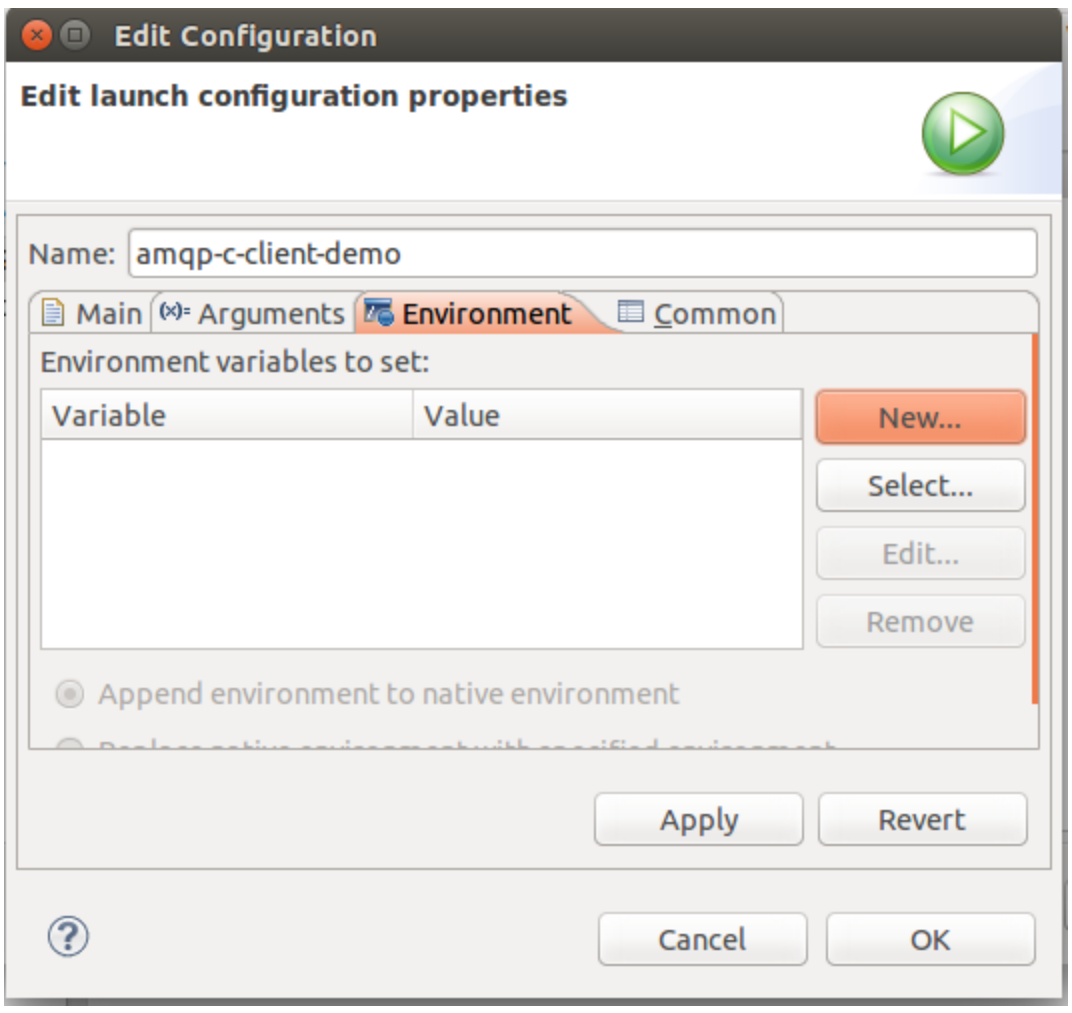
f. Click **OK** to close the **add directory path** dialog.

g. Click **OK** to close the **Properties** window.

7. Create the source file for the client to consume messages.
   a. In **Project Explorer**, right-click **amqp-c-client-demo**, click **New**, and then click **Source File**.
   b. In the **Source** file field, enter the name **amqp_listen_websocket.c**.
   c. In **Template**, ensure that **Default C source template** is selected, and click **Finish**.
8. Add the C code below to **amqp_listen_websocket.c**. The code will create a connection to the Gateway, open a channel with the AMQP broker and log in, consume messages from the broker, and close the close the connection. For the code, see amqp_listen_websocket.c below.
9. Create the source file for the client to produce messages.
   a. In **Project Explorer**, right-click **amqp-c-client-demo**, click **New**, and then click **Source File**.
   b. In the **Source** file field, enter the name **amqp_sendstring_websocket.c**.
   c. In **Template**, ensure that **Default C source template** is selected, and click **Finish**.
10. Add the C code below to **amqp_sendstring_websocket.c**. The code will create a connection to the Gateway, open and bind to a channel with the AMQP broker and log in, produce messages from the broker, and close the connection. For the code, see amqp_sendstring_websocket.c below.
11. From the **File** menu, click **Save All**.
12. Build the Eclipse project.
    a. Right-click the **CDemo** project, and then click **Build Project**.
13. Configure the launch configuration for the AMQP C client.
    a. In the **Properties** window for the project, click the **Run/Debug Settings** heading.

b. Click **New**.

c. In the **Select Configuration Type** dialog, click **C/C++ Application**, and then click **OK**. The **Edit Configuration** window appears.



d. In the **Name** field, enter **amqp-c-client-demo**.

e. Click the **Environment** tab.

f. Click **New** to create a new variable. The **New Environment Variable** window appears.

g. In **Name**, enter **LD_LIBRARY_PATH**.

h. In **Value**, enter the path to the shared object files:

GATEWAY_HOME/lib/client/c/amqp/OS/lib

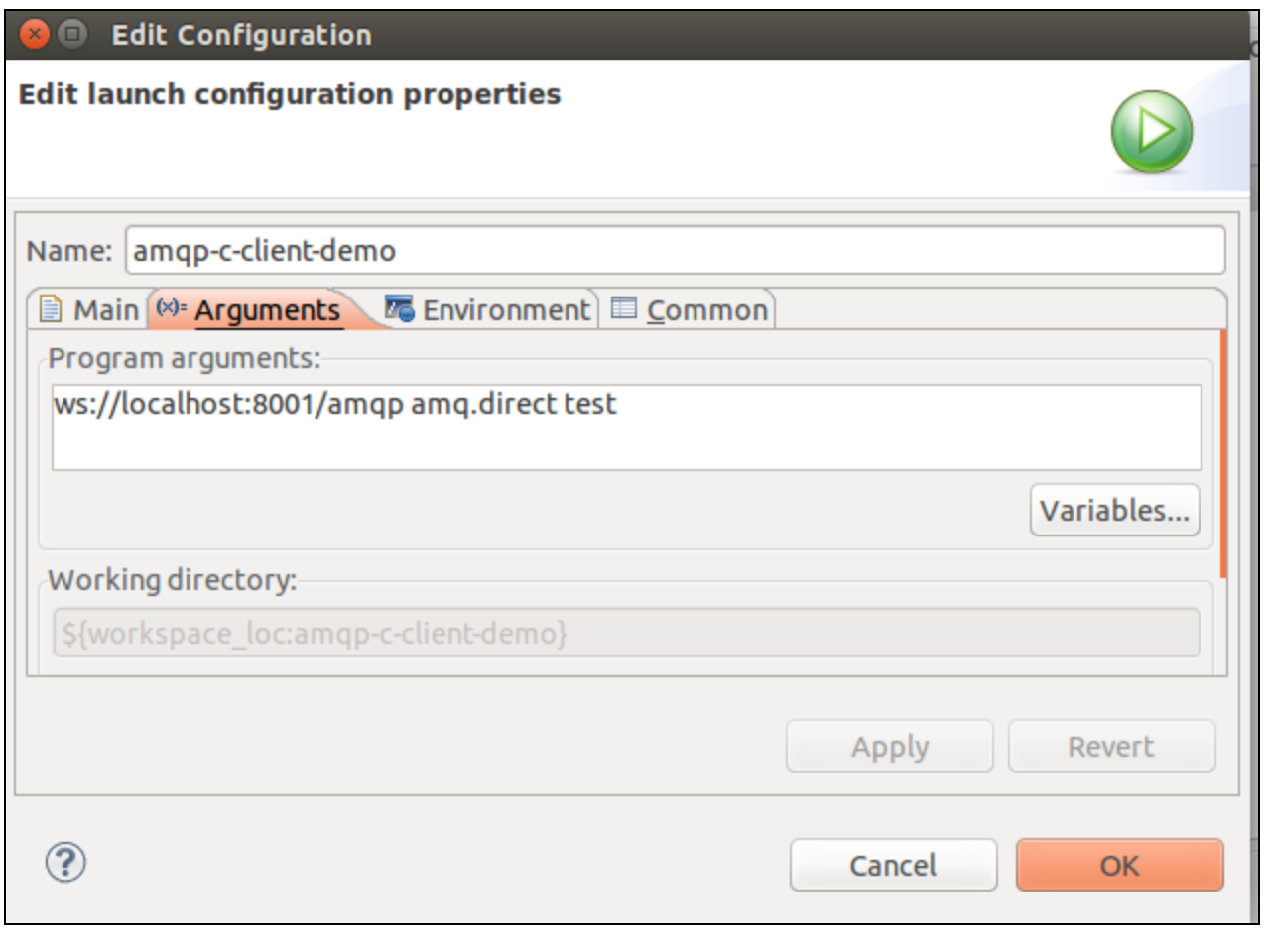
GATEWAY_HOME/lib/client/c/websocket/Ubuntu_precise/lib

i. Click **OK**.

j. Click **OK** to exit the **Edit Configuration** window.

k. Click the **Arguments** tab.

l. Enter the following argument: `ws://localhost:8001/amqp amq.direct test`



m. Click **OK**.

n. Click **OK** to close the **Properties** window.

14. Start the AMQP broker as described in **How do I start Apache Qpid?** in Setting Up Kaazing WebSocket Gateway.
15. Start the Gateway as described in **How do I start and stop the Gateway?** in Setting Up Kaazing WebSocket Gateway.
16. Run your AMQP C client program. In Eclipse, right-click the **amqp-c-client-demo** project, and click **Run**. The client will perform all of its functions in the Console.

Examples

The following examples demonstrate how to:

1. Import the libraries.
2. Create the WebSocket connection to the Gateway.
3. Connect and log into an AMQP broker.
4. Create channels.
5. Declare an exchange.
6. Declare a queue.
7. Bind an exchange to a queue.
8. Consume messages.
9. Publish messages.
10. Handle Exceptions.
11. Use OpenSSL with your C client.

There are four examples:

- amqp_listen_websocket.c
- amqp_sendstring_websocket.c
- amqp_sendstring_websocket_ssl.c
- amqp_sendstring_websocket_ssl_callback.c

amqp_listen_websocket.c

The following example demonstrates how to consume AMQP messages.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <stdint.h>
#include <amqp_websocket.h>
#include <amqp.h>
#include <amqp_framing.h>

#include <assert.h>

#include "utils.h"

int main(int argc, char const *const *argv)
{
  char const *url;
  int status;
  char const *exchange;
  char const *bindingkey;
  amqp_socket_t *socket = NULL;
  amqp_connection_state_t conn;

  amqp_bytes_t queuename;

  if (argc < 4) {
    fprintf(stderr, "Usage: amqp_listen_websocket url exchange bindingkey\n");
    fprintf(stderr, "Example: amqp_sendstring_websocket ws://localhost:8001/amqp amq.direct test\n");
    return 1;
  }

  url = argv[1];
  exchange = argv[2];
```

```
bindingkey = argv[3];

/* Initialize AMQP connection object */
conn = amqp_new_connection();

/*
 * Initialize underlying transport object
 * We are using WebSocket as a transport protocol for AMQP messaging
 */
socket = amqp_websocket_new(conn);
if (!socket) {
  die("creating WebSocket");
}

/* Establish WebSocket connection */
status = amqp_websocket_open(socket, url);
if (status) {
  die("opening WebSocket connection");
}
  /* Establish AMQP connection against the backend broker */
  die_on_amqp_error(amqp_login(conn, "/", 0, AMQP_DEFAULT_FRAME_SIZE, 0, AMQP_SASL_METHOD_PLAIN, "guest",
"guest"),
                    "Logging in");

  /* Open Channel
   * To get the status of call to open channel, amqp_get_rpc_reply should be used
   * amqp_get_rpc_reply() returns the most recent amqp_rpc_reply_t instance corresponding
   * to such an API operation for the given connection.
   */
  amqp_channel_open(conn, 1);
  die_on_amqp_error(amqp_get_rpc_reply(conn), "Opening channel");

  {
     /* Declare Queue
      * In this case we are providing empty name that results in server creating a unique
      * queue name and sending it to the client
      */
    amqp_queue_declare_ok_t *r = amqp_queue_declare(conn, 1, amqp_empty_bytes, 0, 0, 0, 1,
                                 amqp_empty_table);
    die_on_amqp_error(amqp_get_rpc_reply(conn), "Declaring queue");

    /* Get the queue name sent by the server */
    queuename = amqp_bytes_malloc_dup(r->queue);
    if (queuename.bytes == NULL) {
      fprintf(stderr, "Out of memory while copying queue name");
      return 1;
    }
  }
}

/* Bind queue to an exchange */
amqp_queue_bind(conn, 1, queuename, amqp_cstring_bytes(exchange), amqp_cstring_bytes(bindingkey),
                amqp_empty_table);
die_on_amqp_error(amqp_get_rpc_reply(conn), "Binding queue");

/* Start a queue consumer.
 * This method asks the server to start a "consumer", which is a transient request
 * for messages from a specific queue.
 */
amqp_basic_consume(conn, 1, queuename, amqp_empty_bytes, 0, 1, 0, amqp_empty_table);
die_on_amqp_error(amqp_get_rpc_reply(conn), "Consuming");

{
  while (1) {
    amqp_rpc_reply_t res;
```

```
amqp_envelope_t envelope;

amqp_maybe_release_buffers(conn);

/* Wait for and consume a message
 * The function waits for a basic.deliver method on any channel, upon receipt of
 * basic.deliver it reads that message, and returns.
 */
res = amqp_consume_message(conn, &envelope, NULL, 0);

if (AMQP_RESPONSE_NORMAL != res.reply_type) {
  break;
}

printf("Delivery %u, exchange %.*s routingkey %.*s\n",
       (unsigned) envelope.delivery_tag,
       (int) envelope.exchange.len, (char *) envelope.exchange.bytes,
       (int) envelope.routing_key.len, (char *) envelope.routing_key.bytes);

if (envelope.message.properties._flags & AMQP_BASIC_CONTENT_TYPE_FLAG) {
        printf("Content-type: %.*s\n",
               (int) envelope.message.properties.content_type.len,
               (char *) envelope.message.properties.content_type.bytes);
      }

      printf("Message: %.*s\n",
                    (int) envelope.message.body.len,
                    (char *) envelope.message.body.bytes);

      amqp_destroy_envelope(&envelope);
    }
  }

  die_on_amqp_error(amqp_channel_close(conn, 1, AMQP_REPLY_SUCCESS), "Closing channel");
  die_on_amqp_error(amqp_connection_close(conn, AMQP_REPLY_SUCCESS), "Closing connection");
  die_on_error(amqp_destroy_connection(conn), "Ending connection");

  return 0;
}
```

amqp_sendstring_websocket.c

The following example demonstrates how to produce AMQP messages.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <stdint.h>
#include <amqp_websocket.h>
#include <amqp.h>
#include <amqp_framing.h>

#include "utils.h"

int main(int argc, char const *const *argv)
{
  char const *url;
  int status;
  char const *exchange;
  char const *routingkey;
  char const *messagebody;
  amqp_socket_t *socket = NULL;
  amqp_connection_state_t conn;

  if (argc < 5) {
```

```
    fprintf(stderr, "Usage: amqp_sendstring_websocket url exchange routingkey messagebody\n");
    fprintf(stderr, "Example: amqp_sendstring_websocket ws://localhost:8001/amqp amq.direct test \"Hello World\"
\n");
    return 1;
  }

  url = argv[1];
  exchange = argv[2];
  routingkey = argv[3];
  messagebody = argv[4];

  /* Initialize AMQP connection object */
  conn = amqp_new_connection();

  /*
   * Initialize underlying transport object
   * We are using WebSocket as a transport protocol for AMQP messaging
   */
  socket = amqp_websocket_new(conn);
  if (!socket) {
    die("creating WebSocket");
  }

  /* Establish WebSocket connection */
  status = amqp_websocket_open(socket, url);
  if (status) {
    die("opening WebSocket connection");
  }

  /* Establish AMQP connection against the backend broker */
  die_on_amqp_error(amqp_login(conn, "/", 0, 131072, 0, AMQP_SASL_METHOD_PLAIN, "guest", "guest"),

  /* Open Channel
   * To get the status of call to open channel, amqp_get_rpc_reply should be used
   * amqp_get_rpc_reply() returns the most recent amqp_rpc_reply_t instance corresponding
   * to such an API operation for the given connection.
   */                   "Logging in");
  amqp_channel_open(conn, 1);
  die_on_amqp_error(amqp_get_rpc_reply(conn), "Opening channel");

  {
    amqp_basic_properties_t props;
    props._flags = AMQP_BASIC_CONTENT_TYPE_FLAG | AMQP_BASIC_DELIVERY_MODE_FLAG;
    props.content_type = amqp_cstring_bytes("text/plain");
    props.delivery_mode = 2; /* persistent delivery mode */

    /* Publish a message to the broker on an exchange with a routing key. */
    die_on_error(amqp_basic_publish(conn,
                                    1,
                                    amqp_cstring_bytes(exchange),
                                    amqp_cstring_bytes(routingkey),
                                    0,
                                    0,
                                    &props,
                                    amqp_cstring_bytes(messagebody)),
                 "Publishing");
  }

  die_on_amqp_error(amqp_channel_close(conn, 1, AMQP_REPLY_SUCCESS), "Closing channel");
  die_on_amqp_error(amqp_connection_close(conn, AMQP_REPLY_SUCCESS), "Closing connection");
  die_on_error(amqp_destroy_connection(conn), "Ending connection");
  return 0;
}
```

amqp_sendstring_websocket_ssl.c

The following example demonstrates how to produce AMQP messages and verify an OpenSSL client certificate against a Certificate Authority (CA).

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <stdint.h>
#include <amqp_websocket.h>
#include <amqp.h>
#include <amqp_framing.h>

#include "utils.h"

int main(int argc, char const *const *argv)
{
  char const *url;
  int status;
  char const *exchange;
  char const *routingkey;
  char const *messagebody;
  amqp_socket_t *socket = NULL;
  amqp_connection_state_t conn;


  if (argc < 5) {
    fprintf(stderr, "Usage: amqp_sendstring_websocket url exchange routingkey messagebody\n");
    fprintf(stderr, "Example: amqp_sendstring_websocket wss://localhost:9001/amqp amq.direct test \"Hello
World\" \n");
    return 1;
  }

  url = argv[1];
  exchange = argv[2];
  routingkey = argv[3];
  messagebody = argv[4];

  /* Initialize AMQP connection object */
  conn = amqp_new_connection();

  /*
   * Initialize underlying transport object
   * We are using WebSocket as a transport protocol for AMQP messaging
   */
  socket = amqp_websocket_new(conn);
  if (!socket) {
    die("creating WebSocket");
  }

  // Set Client key and certificate
  const char* cert = "/home/user/cert/client.crt";
  const char* key = "/home/user/cert/client.key";
  kws_websocket_t * ws = amqp_websocket_get(socket);
  int setkey = kws_ssl_set_clientkey(ws, cert, key);



  /* Establish WebSocket connection */
  status = amqp_websocket_open(socket, url);
  if (status) {
    die("opening WebSocket connection");
  }

  /* Establish AMQP connection against the backend broker */
  die_on_amqp_error(amqp_login(conn, "/", 0, 131072, 0, AMQP_SASL_METHOD_PLAIN, "guest", "guest"),
```

```
/* Open Channel
 * To get the status of call to open channel, amqp_get_rpc_reply should be used
 * amqp_get_rpc_reply() returns the most recent amqp_rpc_reply_t instance corresponding
 * to such an API operation for the given connection.
 */                  "Logging in");
amqp_channel_open(conn, 1);
die_on_amqp_error(amqp_get_rpc_reply(conn), "Opening channel");

{
  amqp_basic_properties_t props;
  props._flags = AMQP_BASIC_CONTENT_TYPE_FLAG | AMQP_BASIC_DELIVERY_MODE_FLAG;
  props.content_type = amqp_cstring_bytes("text/plain");
  props.delivery_mode = 2; /* persistent delivery mode */

  /* Publish a message to the broker on an exchange with a routing key. */
  die_on_error(amqp_basic_publish(conn,
                                  1,
                                  amqp_cstring_bytes(exchange),
                                  amqp_cstring_bytes(routingkey),
                                  0,
                                  0,
                                  &props,
                                  amqp_cstring_bytes(messagebody)),
               "Publishing");
  }

  die_on_amqp_error(amqp_channel_close(conn, 1, AMQP_REPLY_SUCCESS), "Closing channel");
  die_on_amqp_error(amqp_connection_close(conn, AMQP_REPLY_SUCCESS), "Closing connection");
  die_on_error(amqp_destroy_connection(conn), "Ending connection");
  return 0;
}
```

amqp_sendstring_websocket_ssl_callback.c

The following example demonstrates how to produce AMQP messages and verify an OpenSSL client certificate against a local private key for client authentication.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <stdint.h>
#include <amqp_websocket.h>
#include <amqp.h>
#include <amqp_framing.h>
```

```
#include "utils.h"

static int verify_callback_client_cert(int preverify_ok, X509_STORE_CTX *ctx)
 {

    /*
     * Retrieve the pointer to the SSL of the connection
     */
    SSL* ssl = (SSL*)X509_STORE_CTX_get_ex_data(ctx, SSL_get_ex_data_X509_STORE_CTX_idx());
    //load client certification and private key for client authentication
    int ret;
    ret = SSL_use_certificate_file(ssl, "/home/user/cert/client.crt", SSL_FILETYPE_PEM);
     ret = SSL_use_PrivateKey_file(ssl,"/home/user/cert/client.key", SSL_FILETYPE_PEM);

    return 1;
 }

int main(int argc, char const *const *argv)
{
```

```
  char const *url;
  int status;
  char const *exchange;
  char const *routingkey;
  char const *messagebody;
  amqp_socket_t *socket = NULL;
  amqp_connection_state_t conn;


  if (argc < 5) {
    fprintf(stderr, "Usage: amqp_sendstring_websocket url exchange routingkey messagebody\n");
    fprintf(stderr, "Example: amqp_sendstring_websocket wss://localhost:9001/amqp amq.direct test \"Hello
World\" \n");
    return 1;
  }

  url = argv[1];
  exchange = argv[2];
  routingkey = argv[3];
  messagebody = argv[4];
```

```
/* Initialize AMQP connection object */
conn = amqp_new_connection();

/*
 * Initialize underlying transport object
 * We are using WebSocket as a transport protocol for AMQP messaging
 */
socket = amqp_websocket_new(conn);
if (!socket) {
  die("creating WebSocket");
}

/*
  const char* cert = "/home/user/cert/client.crt";
   const char* key = "/home/user/cert/client.key";
   kws_websocket_t * ws = amqp_websocket_get(socket);
   int setkey = kws_ssl_set_clientkey(ws, cert, key);
   */
kws_websocket_t * ws = amqp_websocket_get(socket);
kws_ssl_set_verify_callback(ws, &verify_callback_client_cert);

/* Establish WebSocket connection */
status = amqp_websocket_open(socket, url);
if (status) {
  die("opening WebSocket connection");
}

/* Establish AMQP connection against the backend broker */
die_on_amqp_error(amqp_login(conn, "/", 0, 131072, 0, AMQP_SASL_METHOD_PLAIN, "guest", "guest"),

/* Open Channel
 * To get the status of call to open channel, amqp_get_rpc_reply should be used
 * amqp_get_rpc_reply() returns the most recent amqp_rpc_reply_t instance corresponding
 * to such an API operation for the given connection.
 */                   "Logging in");
amqp_channel_open(conn, 1);
die_on_amqp_error(amqp_get_rpc_reply(conn), "Opening channel");

{
  amqp_basic_properties_t props;
  props._flags = AMQP_BASIC_CONTENT_TYPE_FLAG | AMQP_BASIC_DELIVERY_MODE_FLAG;
  props.content_type = amqp_cstring_bytes("text/plain");
```

```
    props.delivery_mode = 2; /* persistent delivery mode */

    /* Publish a message to the broker on an exchange with a routing key. */
    die_on_error(amqp_basic_publish(conn,
                                    1,
                                    amqp_cstring_bytes(exchange),
                                    amqp_cstring_bytes(routingkey),
                                    0,
                                    0,
                                    &props,
                                    amqp_cstring_bytes(messagebody)),
                 "Publishing");
  }

  die_on_amqp_error(amqp_channel_close(conn, 1, AMQP_REPLY_SUCCESS), "Closing channel");
  die_on_amqp_error(amqp_connection_close(conn, AMQP_REPLY_SUCCESS), "Closing connection");
  die_on_error(amqp_destroy_connection(conn), "Ending connection");
  return 0;
}
```

Note:

If you select to use this method, then call the `kws_ssl_set_cacert()` function to ensure that you are connected to the correct server. Without setting the CA certificate, the client will connect to the server without any certificate verification. This might cause security problems. Here is the function:

```
int kws_ssl_set_cacert(kws_websocket_t *ws, const char *cacert);
```

The function takes s path to the CA certificate file as a second parameter. The CA certificate file should be of PEM format.

Next Step

Require Clients to Provide Certificates to the Gateway