Medici Extractor in C++

C++

Main Function

```
int main(int argc, const char *argv[])
{
    // need an ip
    if (argc != 2) {
        // report error
        std::cerr << "usage: " << argv[0] << " <ip>" << std::endl;

        // done
        return -1;
    } else {
        // create connection
        MyConnection connection(argv[1]);

        // start the main event loop
        Event::MainLoop::instance()->run();

        // done
        return 0;
    }
}
```

RabbitMQ Connection and Error Handling

```
MyConnection::MyConnection(const std::string &ip) :
    _socket(Event::MainLoop::instance(), this),
    _connection(nullptr),
    _channel(nullptr)
{
    // start connecting
    if (_socket.connect(Network::Ipv4Address(ip), 5672)) return;

    // failure
    onFailure(&_socket);
}


/**
 *  Destructor
 */
MyConnection::~MyConnection()
{
    // do we still have a channel?
    if (_channel)  delete _channel;

    // do we still have a connection?
    if (_connection) delete _connection;
}


/**
 *  Method that is called when the connection failed
 *  @param  socket      Pointer to the socket
 */
void MyConnection::onFailure(Network::TcpSocket *socket)
{
    // report error
    std::cout << "connect failure" << std::endl;
}
```

```
/**
 *  Method that is called when the connection timed out (which also is a failure
 *  @param  socket      Pointer to the socket
 */
void MyConnection::onTimeout(Network::TcpSocket *socket)
{
    // report error
    std::cout << "connect timeout" << std::endl;
}


/**
 *  Method that is called when the connection succeeded
 *  @param  socket      Pointer to the socket
 */
void MyConnection::onConnected(Network::TcpSocket *socket)
{
    // report connection
    std::cout << "connected" << std::endl;

    // we are connected, leap out if there already is a amqp connection
    if (_connection) return;

    // create amqp connection, and a new channel
    _connection = new AMQP::Connection(this, AMQP::Login("guest", "guest"), "/");
    _channel = new AMQP::Channel(_connection, this);

    // we declare a queue, an exchange and we publish a message
    _channel->declareQueue("my_queue");
    _channel->declareExchange("my_exchange", AMQP::direct);
    _channel->bindQueue("my_exchange", "my_queue", "key");
}


/**
 *  Method that is called when the socket is closed (as a result of a TcpSocket::close() call)
 *  @param  socket      Pointer to the socket
 */
void MyConnection::onClosed(Network::TcpSocket *socket)
{
    // show
    std::cout << "myconnection closed" << std::endl;


    // close the channel and connection
    if (_channel) delete _channel;
    if (_connection) delete _connection;

    // set to null
    _channel = nullptr;
    _connection = nullptr;
}


/**
 *  Method that is called when the peer closed the connection
 *  @param  socket      Pointer to the socket
 */
void MyConnection::onLost(Network::TcpSocket *socket)
{
    // report error
    std::cout << "connection lost" << std::endl;

    // close the channel and connection
    if (_channel) delete _channel;
    if (_connection) delete _connection;

    // set to null
    _channel = nullptr;
    _connection = nullptr;
}
```

```
/**
 *  Method that is called when data is received on the socket
 *  @param  socket      Pointer to the socket
 *  @param  buffer      Pointer to the fill input buffer
 */
void MyConnection::onData(Network::TcpSocket *socket, Network::Buffer *buffer)
{
    // send what came in
    std::cout << "received: " << buffer->size() << " bytes" << std::endl;

    // leap out if there is no connection
    if (!_connection) return;

    // let the data be handled by the connection
    size_t bytes = _connection->parse(buffer->data(), buffer->size());

    // shrink the buffer
    buffer->shrink(bytes);
}


/**
 *  Method that is called when data needs to be sent over the network
 *
 *  Note that the AMQP library does no buffering by itself. This means
 *  that this method should always send out all data or do the buffering
 *  itself.
 *
 *  @param  connection      The connection that created this output
 *  @param  buffer          Data to send
 *  @param  size            Size of the buffer
 */
void MyConnection::onData(AMQP::Connection *connection, const char *buffer, size_t size)
{
    // send to the socket
    _socket.write(buffer, size);
}


/**
 *  When the connection ends up in an error state this method is called.
 *  This happens when data comes in that does not match the AMQP protocol
 *
 *  After this method is called, the connection no longer is in a valid
 *  state and can be used. In normal circumstances this method is not called.
 *
 *  @param  connection      The connection that entered the error state
 *  @param  message         Error message
 */
void MyConnection::onError(AMQP::Connection *connection, const std::string &message)
{
    // report error
    std::cout << "AMQP Connection error: " << message << std::endl;
}


/**
 *  Method that is called when the login attempt succeeded. After this method
 *  was called, the connection is ready to use
 *
 *  @param  connection      The connection that can now be used
 */
void MyConnection::onConnected(AMQP::Connection *connection)
{
    // show
    std::cout << "AMQP login success" << std::endl;

    // create channel if it does not yet exist
    if (!_channel) _channel = new AMQP::Channel(connection, this);
```

```
}


/**
 *  Method that is called when the channel was succesfully created.
 *  Only after the channel was created, you can use it for subsequent messages over it
 *  @param  channel
 */
void MyConnection::onReady(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP channel ready, id: " << (int) channel->id() << std::endl;
}


/**
 *  An error has occured on the channel
 *  @param  channel
 *  @param  message
 */
void MyConnection::onError(AMQP::Channel *channel, const std::string &message)
{
    // show
    std::cout << "AMQP channel error, id: " << (int) channel->id() << " - message: " << message << std::endl;


    // main channel cause an error, get rid of if
    delete _channel;


    // reset pointer
    _channel = nullptr;
}

/**
 *  Method that is called when the channel was paused
 *  @param  channel
 */
void MyConnection::onPaused(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP channel paused" << std::endl;
}


/**
 *  Method that is called when the channel was resumed
 *  @param  channel
 */
void MyConnection::onResumed(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP channel resumed" << std::endl;
}


/**
 *  Method that is called when a channel is closed
 *  @param  channel
 */
void MyConnection::onClosed(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP channel closed" << std::endl;
}


/**
 *  Method that is called when a transaction was started
 *  @param  channel
 */
```

```
void MyConnection::onTransactionStarted(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP transaction started" << std::endl;
}


/**
 *  Method that is called when a transaction was committed
 *  @param  channel
 */
void MyConnection::onTransactionCommitted(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP transaction committed" << std::endl;
}


/**
 *  Method that is called when a transaction was rolled back
 *  @param  channel
 */
void MyConnection::onTransactionRolledBack(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP transaction rolled back" << std::endl;
}


/**
 *  Mehod that is called when an exchange is declared
 *  @param  channel
 */
void MyConnection::onExchangeDeclared(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP exchange declared" << std::endl;
}


/**
 *  Method that is called when an exchange is bound
 *  @param  channel
 */
void MyConnection::onExchangeBound(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP Exchange bound" << std::endl;
}


/**
 *  Method that is called when an exchange is unbound
 *  @param  channel
 */
void MyConnection::onExchangeUnbound(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP Exchange unbound" << std::endl;
}


/**
 *  Method that is called when an exchange is deleted
 *  @param  channel
 */
void MyConnection::onExchangeDeleted(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP Exchange deleted" << std::endl;
}
```

```
/**
 *  Method that is called when a queue is declared
 *  @param  channel
 *  @param  name            name of the queue
 *  @param  messageCount    number of messages in queue
 *  @param  consumerCount   number of active consumers
 */
void MyConnection::onQueueDeclared(AMQP::Channel *channel, const std::string &name, uint32_t messageCount,
uint32_t consumerCount)
{
    // show
    std::cout << "AMQP Queue declared" << std::endl;
}


/**
 *  Method that is called when a queue is bound
 *  @param  channel
 *  @param
 */
void MyConnection::onQueueBound(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP Queue bound" << std::endl;

    _channel->publish("my_exchange", "invalid-key", AMQP::mandatory, "this is the message");
}


/**
 *  Method that is called when a queue is deleted
 *  @param  channel
 *  @param  messageCount    number of messages deleted along with the queue
 */
void MyConnection::onQueueDeleted(AMQP::Channel *channel, uint32_t messageCount)
{
    // show
    std::cout << "AMQP Queue deleted" << std::endl;
}


/**
 *  Method that is called when a queue is unbound
 *  @param  channel
 */
void MyConnection::onQueueUnbound(AMQP::Channel *channel)
{
    // show
    std::cout << "AMQP Queue unbound" << std::endl;
}


/**
 *  Method that is called when a queue is purged
 *  @param  messageCount        number of message purged
 */
void MyConnection::onQueuePurged(AMQP::Channel *channel, uint32_t messageCount)
{
    // show
    std::cout << "AMQP Queue purged" << std::endl;
}


/**
 *  Method that is called when the quality-of-service was changed
 *  This is the result of a call to Channel::setQos()
 */
void MyConnection::onQosSet(AMQP::Channel *channel)
{
```

```
    // show
    std::cout << "AMQP Qos set" << std::endl;
}


/**
 *  Method that is called when a consumer was started
 *  This is the result of a call to Channel::consume()
 *  @param  channel         the channel on which the consumer was started
 *  @param  tag             the consumer tag
 */
void MyConnection::onConsumerStarted(AMQP::Channel *channel, const std::string &tag)
{
    // show
    std::cout << "AMQP consumer started" << std::endl;
}


/**
 *  Method that is called when a message has been received on a channel
 *  @param  channel         the channel on which the consumer was started
 *  @param  message         the consumed message
 *  @param  deliveryTag     the delivery tag, you need this to acknowledge the message
 *  @param  consumerTag     the consumer identifier that was used to retrieve this message
 *  @param  redelivered     is this a redelivered message?
 */
void MyConnection::onReceived(AMQP::Channel *channel, const AMQP::Message &message, uint64_t deliveryTag, const
std::string &consumerTag, bool redelivered)
{
    // show
    std::cout << "AMQP consumed: " << message.message() << std::endl;

    // ack the message
    channel->ack(deliveryTag);
}


/**
 *  Method that is called when a message you tried to publish was returned
 *  by the server. This only happens when the 'mandatory' or 'immediate' flag
 *  was set with the Channel::publish() call.
 *  @param  channel         the channel on which the message was returned
 *  @param  message         the returned message
 *  @param  code            the reply code
 *  @param  text            human readable reply reason
 */
void MyConnection::onReturned(AMQP::Channel *channel, const AMQP::Message &message, int16_t code, const std::
string &text)
{
    // show
    std::cout << "AMQP message returned: " << text << std::endl;
}


/**
 *  Method that is called when a consumer was stopped
 *  This is the result of a call to Channel::cancel()
 *  @param  channel         the channel on which the consumer was stopped
 *  @param  tag             the consumer tag
 */
void MyConnection::onConsumerStopped(AMQP::Channel *channel, const std::string &tag)
{
    // show
    std::cout << "AMQP consumer stopped" << std::endl;
}
```