# MTS SDK Java Developer Guide

July 2025

betradar
driven by facts

## Table of Contents

# 1 Introduction

To make **M**anaged **T**rading **S**ervice (MTS) integration as quick and easy as possible **S**oftware **D**evelopment **K**it (SDK) was developed.

SDK exposes MTS ticket interface in a more user-friendly way and isolates the client from having to do proper connection handling, throttling, message parsing and sending. It also adds valuable diagnostics information, which in turn can help to solve client support issues.

This document contains info about Java implementation and usage of the SDK.

# 2 SDK High Level Overview

SDK under the hood is working over either AMQP or WebSocket protocols. While the SDK tries to abstract away as much details as possible, clients should still be familiar with basics of the AMQP or WebSocket protocols.

AMQP specifications are available at https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf

SDK will be always built against the latest version of the MTS ticket protocol so that a client can simply choose to upgrade the protocol (SDK) whenever he wants to and get all the benefits of new functionality immediately. We still recommend testing each new version of the SDK against the MTS staging (i.e. client integration) environment.

With each new release we will provide release notes where it will be stated if there are any breaking changes in the SDK interface or the protocol itself and what are all the new features and bug fixes.

For further information we also recommend reading “MTS integration with AMQP” and “MTS Ticket Integration” documents, which you should have already received during the course of integration.

## 3 Start using the SDK

In this document pseudo code is used to give you some basic understanding. For actual working examples please take a look at the accompanied SDK example project.

First create a new instance of MTS SDK

```
MtsSdkApi mtsSdk = new MtsSdk();
```

This creates the SDK but does not start it yet so no connections are established at this point.

To actually initiate a connection with MTS RabbitMQ or WebSocket ticket gateway, you have to open the connection first. There are three possible ways.

First option:

```
MtsSdkApi.open()
```

This will initialize the SDK using the “mts-sdk.properties” file located in your resources folder.

Second option:

```
MtsSdkApi.open(String filePath)
```

This will initialize the SDK using the file found in the path you specified. It can be either an absolute or relative path. Settings format in the file must be the same as in “mts-sdk.properties”.

Third option:

```
MtsSdkApi.open(Properties properties)
```

This will initialize the SDK using provided properties where key-value pairs correspond to contents of the “mts-sdk.properties” file.

MTS SDK connects over AMQP by default, unless you configure the properties to use WebSocket connection instead (see configuration).

It is possible to create multiple SDK instances using different settings, which can prove useful if you want to run multiple SDKs on the same JVM, thus connecting through multiple RabbitMQ virtual hosts or WebSockets.

After SDK is initialized you can create one or more messages senders. There are four different types of senders based on the actual use case:

- TicketSender
- TicketCancelSender
- TicketAcknowledgmentSender
- *TicketCancelAcknowledgmentSender*

## 3.1 Ticket Sender

The main sender is TicketSender which is used to send tickets to MTS and receive acceptance information in response. You obtain a new instance by calling

```
TicketSender ticketSender =
MtsSdkApi.getTicketSender(TicketResponseListener responseListener)
```

You have to supply your implementation of response handler where you will receive events on whether ticket was successfully published or not (i.e. AMQP publisher confirms) as well as MTS acceptance response, i.e. whether ticket should be accepted or rejected by the client.

There are two ways of sending tickets. The preferred way is asynchronously but you can also send them synchronously. First one is by using

```
TicketSender.send(Ticket ticket)
```

This will send a ticket asynchronously and trigger response in your response listener. This is the recommended way of sending tickets.

Clients who find synchronous interface better suiting to their needs can send tickets in a blocking fashion

```
TicketResponse response = TicketSender.sendBlocking(Ticket ticket) throws
ResponseTimeoutException
```

This method will block calling thread until ticket is either accepted or rejected or throw ResponseTimeoutException if reply is not received in time (usually after 15s). This method returns a TicketResponse. TicketResponseListener will not be triggered in this case. The timeout can be set using the ticketResponseTimeout configuration attribute.

Before sending any ticket you need to generate a Ticket using TicketBuilder. MTS ticket is structured hierarchically but we made the builder flat with only selections as sub objects of the ticket. Below is an example of how a ticket might be constructed.

```
Ticket ticket =  TicketSender.newBuilder()
        .setBookmakerId(10)
        .setTicketId("f2f5b035-7ded-4527-9c3f-73cab71fb15b")
        .setLimitId(2)
        .setChannelId(SourceChannel.INTERNET)
        .setDeviceId("e4fe9bde-caa0-47b6-908d-ffba3fa184f2")
        .setEndCustomerId("User123456")
        .setEndCustomerIp("1.3.3.7")
        .setLanguageId("EN")
        .setCurrency("EUR")
        .setStake(5.0)
        .setSystem(0) // accumulator bet
        .setBonusWin(10.2)
        .addSelection()
          .setLine(LineType.PREMATCH)
          .setMarket("lcoo:10/1/*/1")
          .setMatch(9569629)
          .setOdd(1.1)
          .buildSelection()
        .addSelection()
          .setLine(LineType.PREMATCH)
          .setMarket("lcoo:12/2/*/2")
          .setMatch(5369329)
          .setOdd(1.3)
          .buildSelection()
        .build()
```

To add selections you call

```
TicketBuilder.addSelection()
```

It returns selection builder where you can set various selection properties. When you are done with selection you call

```
SelectionBuilder.buildSelection()
```

It builds the selection, adds the selection to parent object and returns the ticket builder.

### 3.2 Ticket Cancellation Sender

If you want to cancel a ticket use TicketCancelSender. Similar to ticket sender both synchronous and asynchronous types of interfaces are supported. Synchronous calls are discouraged because response for ticket cancellation could be send several minutes after the initial request. The timeout for ticket cancellation can be set using ticketCancellationResponseTimeout configuration attribute.

New instance is created by

```
TicketCancelSender ticketSender = MtsSdkApi.getTicketCancelSender(
TicketCancelResponseListener responseListener)
```

*You should supply* ***ResponseListener*** where you will receive notifications of whether ticket cancellation was successfully published and to get a final ticket cancellation response from MTS, i.e. cancellation accepted or rejected.

Example of ticket cancellation

```
TicketCancel ticketCancel = TicketCancelSender.newBuilder()
        .setCancelMessageId("messageID")
        .setTicketId("ticket id")
        .setBookmakerId(1)
        .setCancellationReason(101)
        .setReasonMessage("customer cancelled ticket")
        .build()
```

After you have constructed a TicketCancel object you have to send it with TicketCancelSender

```
TicketCancelSender.send(TicketCancel ticketCancel)
```

### 3.3 Ticket Acknowledgement Sender

Ticket acknowledgment sender can be used to acknowledge back to MTS of whether preceding MTS ticket acceptance suggestion was followed or not by the client which in turn makes it easier to reconcile the records on both sides.

As acknowledgments are one-way you will only be able to receive confirmations of whether acknowledgement was successfully delivered to MTS or not. There will be no other replies.

Acknowledgement sender creation:

```
TicketAcknowledgmentSender ticketAckSender =
MtsSdkApi.getTicketAcknowledgmentSender (
TicketAcknowledgmentResponseListener responseListener)
```

Acknowledgement message creation:

```
TicketAcknowledgment ticketAcknowledgment =
TicketAcknowledgmentSender.newBuilder()
        .setTicketId("ticket id")
        .setAckStatus(TicketAckStatus.ACCEPTED)
        .setBookmakerId(1)
        .setSourceCode(100)
        .build();
```

Acknowledgement sending:

```
TicketAcknowledgmentSender.send(TicketAcknowledgment  ticketAcknowledgment);
```

### 3.4 Ticket Cancel Acknowledgement Sender

Ticket cancellation acknowledgment sender can be used to acknowledge back to MTS of whether preceding MTS ticket cancellation response was followed or not by the client which in turn makes it easier to reconcile the records on both sides.

Sender creation:

```
TicketCancelAcknowledgmentSender ticketCancelAckSender =
MtsSdkApi.getTicketCancelAcknowledgmentSender (
TicketCancelAcknowledgmentResponseListener responseListener)
```

Message creation:

```
TicketCancelAcknowledgment ticketCancelAcknowledgment =
TicketCancelAcknowledgmentSender.newBuilder()
        .setTicketId("ticket id")
        .setAckStatus(TicketCancelAckStatus.CANCELLED)
        .setBookmakerId(1)
        .setSourceCode(101)
        .build();
```

Message sending:

```
TicketCancelAcknowledgmentSender.send(TicketCancelAcknowledgment
ticketCancelAcknowledgment);
```

# 4 SDK Configuration Settings

| Setting | Mandatory | Default | Description |
|---|---|---|---|
| **mts.sdk.username** | yes | | AMQP username |
| **mts.sdk.password** | yes | | AMQP password |
| **mts.sdk.hostname** | yes | | Hostname<br>CI: mtsgate-ci.betradar.com<br>Prod: mtsgate-t1.betradar.com |
| **mts.sdk.vhost** | no | | AMQP virtual host (default: /username) |
| **mts.sdk.ssl** | no | true | Use SSL for communication |
| **mts.sdk.port** | no | 5671 | Port<br>5671 if ssl=true, else 5672 |
| **mts.sdk.node** | no | 1 | Node id to be used when creating routing key |
| **mts.sdk.bookmakerId** | no | 0 | When provided, it is used as the default value for the BookmakerId on the ticket. |
| **mts.sdk.limitId** | no | 0 | When provided, it is used as the default value for the LimitId property on the ticket. |
| **mts.sdk.currency** | no | | When provided, it is used as the default value for the Currency property on the ticket. |
| **mts.sdk.channel** | no | | When provided, it is used as the default value for the<br>SenderChannel property on the ticket. |
| **mts.sdk.accessToken** | no | | When selections are build using UnifiedOdds ids, the accessToken is used to access sports API. |
| **mts.sdk.<br>provideAdditionalMarket<br>Specifiers** | no | true | This value is used to indicate if the sdk should add market specifiers for specific markets. Only used when building selection using UnifiedOdds ids. |
| **mts.sdk.<br>exclusiveConsumer** | no | true | The value specifying whether the rabbit consumer channel should be exclusive. |
| **mts.sdk.keycloakHost** | no | | The auth server for accessing MTS Client API. |
| **mts.sdk.<br>keycloakUsername** | no | | The default username used to get access token from the auth server. It can be overridden when the MTS Client API methods are called. |
| **mts.sdk.<br>keycloakPassword** | no | | The default password used to get access token from the auth server. It can be overridden when the MTS Client API methods are called. |
| **mts.sdk.keycloakSecret** | no | | The secret used to get access token from the auth server. |
| **mts.sdk.<br>mtsClientApiHost** | no | | The MTS Client API host. |
| **mts.sdk.ticketResponse<br>Timeout** | no | 15000 | The ticket response timeout(ms). |
| **mts.sdk.ticketCancellati<br>onResponseTimeout** | no | 600000 | The ticket cancellation response timeout(ms). |

| | | | |
|---|---|---|---|
| **mts.sdk.ticketCashoutResponseTimeout** | no | 600000 | The ticket cashout response timeout(ms). |
| **mts.sdk.ticketNonSrSettleResponseTimeout** | no | 600000 | The ticket non-Sportradar response timeout(ms). |
| **mts.sdk.useWebsocket** | Yes* | False | Use WebSocket connection instead of AMQP |
| **mts.sdk.authServer** | Yes* | | URI of the authorization server to send a token request to |
| **mts.sdk.authClientId** | Yes* | | Client ID to include in the token request |
| **mts.sdk.authClientSecret** | Yes* | | Client secret to include in the token request |
| **mts.sdk.authAudience** | Yes* | | Audience attribute to use in the token request |
| **mts.sdk.wsServer** | Yes* | | URI of the WebSocket server to connect to. |
| **mts.sdk.protocolMaxSendBufferSize** | No | 1000 | Specifies the maximum number of requests that have been sent but whose responses have not yet been received |
| **mts.sdk.protocolReceiveResponseTimeout** | No | 30s | Specifies the maximum time the SDK will wait for response to be received |
| **mts.sdk.protocolRetryCount** | No | 0 | Specifies the number of retries to send a request after the response timeout has expired |

*= mandatory settings if connecting over WebSocket