LD - F1

Introduction

The Sportradar F1 Live data feed is the best way to secure low-latency real-time data from F1 races. It gives you access to leaderboard updates, timing data, information about pit-stops and much more through a gRPC event stream.

Basic usage

This document describes how you can connect to the feed and receive events. It's encouraged to follow the quick start guide and to use the replay functionality in the integration process. This ensures that you'll develop a robust solution in addition to making it easy to test your integration. In this document you will also find basic information about the F1 sport to ease developer understanding of the feed. Triggers for each event in addition to detailed descriptions are also outlined in this document. Finally this document contains a section of code-snippets to make it easy to get started. You should be able to start your first replay in less than one hour after you have secured your SSO token and booked your first replayable stage. It's that easy.

Sportradar F1 live data feed

This F1 Live Data feed is delivered as a gRPC service. That makes it fast, reliable and easy to work with in a broad range of languages. The Code Snippets section serves as an example in Java.

gRPC

gRPC is a high-performance, open-source universal RPC framework where you use protocol buffers to describe your service. From these files you can automatically create client stubs that work in a variety of languages and platforms. It allows the client to call methods directly on the server application on a different machine as it was a local object. On the client side you use a automatically generated stub (aka client) to call methods on the server.

gRPC serializes the data to minimize data transfer and ensure fast communication. The client stubs convert this into objects you can easily work with in your favorite language.

Visit https://grpc.github.io/ to learn more about gRPC

Help

If you have any questions or queries please do not hesitate to contact our support team: support@sportradar.com

For any questions regarding commercial matters, please contact sales@sportradar.com

Access method

The F1 feed is provided as a gRPC service.

Host	stream.ld.betradar.com
Port	443
Proto files	https://github.com/sportradar/livedataf1

Access restrictions

To access the service you need a valid SSO token and to have booked F1 stages. The token is sent with each request as gRPC metadata - "Authorization". A java sample of how to do this is shown in the Code Samples section at the bottom of this document.

Tokens can be generated here: https://ufadmin.betradar.com/

Quick Start

Before you can start you need a SportRadar SSO token and you need to book one or more stages.

The easiest way to get started is to follow one of the code samples at the end of this documentation.

Testing gRPC

For quickly testing gRPC there are several tools you can use. Take a look at the extensive list provided here: https://github.com/grpc-ecosystem/awesome-grpc#tools-cli

Example gRPC client tools are gRPCurl and BloomRPC.

Sample call to stream events fro a given stageId.

gRPCurl sample

```
grpcurl -proto services.proto -d ‘{“stageId”:“<StageId>"}’ -H
‘Authorization: <token-from-sportradar-sso>’ stream.ld.betradar.com:443
sportradar.ldi.f1.services.v1.EventStream/StreamEvents
```

Sample call to replay a given stage with 2x normal speed. StageId used here is from the Belgian GP Race 2019.

gRPCurl sample - ReplayStreamEvents

```
grpcurl -proto services.proto -d '{"stageId":"sr:stage:430547",
"speedFactor": 2}' -H 'Authorization: <token-from-sportradar-sso>'
stream.ld.betradar.com:443 sportradar.ldi.f1.services.v1.EventStream
/ReplayStreamEvents
```

Sample call to get a snapshot of the latest stage of a given stage. StageId used here is from the Belgian GP Race 2019.

gRPCurl sample - GetStageSnapshot

```
grpcurl -proto services.proto -d '{"stageId":"sr:stage:430547"}' -H
'Authorization: <token-from-sportradar-sso>' stream.ld.betradar.com:443
sportradar.ldi.f1.services.v1.StageInfo/GetStageSnapshot
```

Sample call to get details about a specific stage. StageId used here is from the Belgian GP Race 2019.

gRPCurl sample - GetStageDetails

```
grpcurl -proto stage-discovery.proto -d '{"stageId":"sr:stage:430547"}' -
H 'Authorization: <token-from-sportradar-sso>' stream.ld.betradar.com:
443 sportradar.ldi.stage_discovery.v1.StageDiscovery/GetStageDetails
```

Sample call to discover stages with GetStageTimetable. Using the timerange around the Belgian GP Race from 2019 to get that stageId. Note that when using timestamps in gRPCurl you need to use a RFC string as shown in the example below. In the internals of gRPC this will be transformed into a google.protobuf.Timestamp type consisting of seconds and nanos.

gRPCurl sample - GetStageTimetable

```
grpcurl -proto stage-discovery.proto -d '{"sportId":"sr:sport:40",
"from": "2019-08-30T00:00:00Z", "to": "2019-09-02T00:00:00Z"}' -H
'Authorization: <token-from-sportradar-sso>' stream.ld.betradar.com:443
sportradar.ldi.stage_discovery.v1.StageDiscovery/GetStageTimetable
```

Discover stages

The first thing you need to do is to set up your project with the proto files and to secure an SSO token. Once you have that you are able to call the RPC GetStageTimetable. That procedure will return **stage s you have booked** for the requested sport and time-frame. For now request stages with the sport id **sr:sport:40** and a time range around the 2019 Belgian GP; 30 August - 01 September 2019.

From this you'll get a list of available stages that you have booked. If no stages are returned please make sure that you have booked some stages and that they exist in the requested time-range. If needed, expand the time-range.

This response will be in a protocol buffer format. Properties are typed and easy to extract and work with. Most languages make it easy to convert protocol-buffers to json, **for readability reasons we convert the sample responses to json** - to make it possible to share them here.

To replay an event we need one stage where isInProgress is false. Take a note of that stageId and proceed to the next section; Replay Events.

Sample GetStageTimetable response converted to JSON

```
{
    "stages": [
        {
            "stageId": "<stage 1>",
            "startEvents": {
                "seconds": 1582813200,
                "nanos": 0
            },
            "stageStart": {
                "seconds": 1582813800,
                "nanos": 0
            },
            "stageEnd": {
                "seconds": 1582821000,
                "nanos": 0
            },
            "name": "Race",
            "description": "Sample race 1",
            "sportId": "40",
            "categoryId": "36",
            "categoryName": "Formula 1",
            "parentStageName": "Grosser Preis von Deutschland 2019",
            "stageType": "STAGE_TYPE_RACE",
            "infoTypes": [
                {
                    "infoid": "<stage 1>:status",
                    "name": "Status",
                    "value": "Finished",
                    "determinate": "status"
                },
            ],
            "isInProgress": false
        },
        {
            "stageId": "<stage 2>",
            "startEvents": {
                "seconds": 1582892409,
                "nanos": 128000000
            },
            "stageStart": {
                "seconds": 1582893009,
                "nanos": 128000000
            },
            "stageEnd": {
                "seconds": 1582900209,
                "nanos": 128000000
            },
            "name": "Race",
            "description": "Sample race 2",
            "sportId": "40",
            "categoryId": "36",
            "categoryName": "Formula 1",
            "parentStageName": "Grand Prix de France 2019",
            "stageType": "STAGE_TYPE_RACE",
            "infoTypes": [

            ],
            "isInProgress": false
        },
        {
            "stageId": "<stage 3>",
            "startEvents": {
                "seconds": 1583329822,
                "nanos": 738000000
            },
            "stageStart": {
                "seconds": 1583330422,
                "nanos": 738000000
            },
            "stageEnd": {
                "seconds": 1583337622,
                "nanos": 738000000
            },
            "name": "Race",
            "description": "Sample race 3",
            "sportId": "40",
            "categoryId": "36",
            "categoryName": "Formula 1",
            "parentStageName": "Australian Grand Prix 2019",
            "stageType": "STAGE_TYPE_RACE",
            "infoTypes": [
                {
                    "infoid": "<stage 3>:laps",
                    "name": "Rounds",
                    "value": "58",
                    "determinate": "laps"
                }
            ],
            "isInProgress": true
        }
    ]
}
```

Replay Events

We have built this service to make it easy for developers to integrate and test their solution. This means that you can start a replay of any historic stage you have booked when you want. You can replay at your desired speed and you can restart or stop the replay when it suits you.

To get started look up the RPC ReplayStreamEvents. The response from this call will be a stream in exactly the same format as you should expect for live stages, however this is tailor made for integration testing. For that reason this procedure has two differences you need to be aware of.

1. It will disconnect you at random time interval - this is to ensure that you implement a reconnection strategy and that you are able to test that.
2. It mimics real timing for previous stages and allows you to speed up the race with a "fast-forward" parameter - "speedFactor". With this you can replay stages up to 10x the normal speed.

Reconnection strategy

It is crucial that you implement a reconnection strategy from the start. This ensures that you are able to reconnect if some unexpected issue should stop the stream of events. By keeping track of the last event you received you can continue streaming from where you left off. This makes it easy to make your implementation Highly Available and it ensures that you will have a reliable implementation.

When you connect the server will start streaming events from the beginning of the stage. All events will be in the form of an EventResponse with a single EventWrapper property. The event wrapper contains the id of the event (sequence id) and the event it self in addition to some other metadata. All stages start with a StartOfStageEvent wrapped in the event wrapper that indicates that the data-stream for this stage has started. When the stage is finalised you'll get an EndOfStageEvent wrapped in the event wrapper, this indicates that there will be no more events and that you should close the stream. If you don't close the stream the sever will do that at the indicated time. Normally 5 minutes after the EndOfStageEvent is sent.

Event frequency

During a Formula 1 Race stage you should expect to see somewhere around 50-70k events in the span of 2.5hours. Around 8 events/second on average and around 30 events/second peak. Make sure your setup is capable of handling this load.

You will receive a stream of EventResponse. Each of them contains an EventWrapper with an event. These events need to be unpacked/de-serialized. Before you unpack the event it will look like this if converted to JSON.

Sample EventResponse converted to JSON

```
{
    "eventWrapper": {
        "id": 615980,
        "rawEventUuid": "",
        "stageId": "<stage 1>",
        "loggedAt": {
            "seconds": 1579872820,
            "nanos": 0
        },
        "eventType": "WeatherUpdateEvent",
        "event": {
            "typeUrl": "type.googleapis.com/sportradar.ldi.f1.events.v1.
WeatherUpdateEvent",
            "value":
"CWZmZmZmJlZAEAEZAAAAAAAANUAhmpmZmZn5jkApAAAAAAAAOkAwqAE="
        }
    }
}
```

When you unpack the event, in this case a WeatherUpdateEvent, the event will look like this when you convert it to JSON. Take a look at the Code samples section at the end of this document to see how you can do this in Java.

Sample WeatherUpdateEvent converted to JSON

```
{
    "humidity": 88.2,
    "rainfall": true,
    "airTemp": 21,
    "pressure": 991,
    "trackTemp": 26,
    "windDirection": 0,
    "windSpeed": 1.3
}
```

Outline flow

The illustration below shows how you should connect and stream events in four different scenarios:

First: Discover a stage/race by calling GetStageTimetable and getting a list of available stages in a StageTimetableResponse.

1. Replay a recent race. In the timetable response from the first stage you might find a stage that was done 14 days ago. You can replay this when you want by calling ReplayStreamEvents. From that you'll get a stream of EventResponse.
2. Get a recent stage. Instead of replaying a previous stage you can also fetch the data as fast as possible. To do this you can call StreamEvents for that stageId with afterSequenceId 0. That will stream all events from the beginning of the stage as fast as your network connection allows. You can also use StreamEvents with afterSequenceId: 0 for a live stage, in this case you'll get all previous events as fast as possible until you catch up to the most recent one, after you catch up you'll receive new events as soon as they are created.
3. Stream an ongoing race. First get a snapshot of the ongoing stage to get the current state of the stage. In that snapshot you'll get the id of the last event used to generate the snapshot. After you have received the snapshot, start streaming events for the stage with StreamEvents with the afterSequenceId set to the event id from the snapshot. That way you'll continue streaming from the point in time when the snapshot where created (snapshots are created when you request them).
4. Upcoming stage. Connect with StreamEvents at the suggested startEvents time you got from the StageTimetableResponse. If the stage has not started you'll get disconnected with a NOT_FOUND error. If that happens, back-off for 10s and reconnect. Once the stage starts you'll get a stream of EventResponse.

Service overview

The procedures are grouped in 3 services; StageDiscovery, EventStream and StageInfo.

StageDiscovery

StageDiscovery gives you an easy way to discover stages you **have booked.** You can either search in a time range for a given sport, or you can look up details for a specific stage to get information about when the stage starts and when we recommend to connect.

sportId

For Formula 1 you should always **use the sportId sr:sport:40**

Stage

A stage is the same as a sporting event. For Formula 1 this is a part of a Grand Prix. Each Grand Prix consists of 5 stages; Practice 1, Practice 2, Practice 3, Qualifying and Race. For season start 2020 we offer live data from Race stages.

Stage			
Propery	**Type**	**Description**	**Sample value**
stageId	string	The id of the stage. This is used when you want to stream events from a specific stage.	sr:stage: 00001
startEvents	google. protobuf. Timestamp	The estimated time when we start streaming live data from this stage. This is the time when you should connect. Seconds and nanos since Unix epoch, that is the time 00:00:00 **UTC** on 1 January 1970, minus leap seconds	{ seconds: 1582626479, nanos: 385000000 }
stageStart	google. protobuf. Timestamp	The official start time of the stage. Note: F1 might start the stage slightly before or after this time, so be sure to connect at the time indicated in startEvents property. Seconds and nanos since Unix epoch, that is the time 00:00:00 **UTC** on 1 January 1970, minus leap seconds	{ seconds: 1582626479, nanos: 385000000 }
stageEnd	google. protobuf. Timestamp	The official end time of the stage. Duration for F1 Race stages are normally around 2 hours. The data-stream lasts a bit longer than this, but normally not much. We will send an EndOfStageEvent to let you know when the stage is done. At that time you should disconnect from the stream. Seconds and nanos since Unix epoch, that is the time 00:00:00 **UTC** on 1 January 1970, minus leap seconds	{ seconds: 1582626479, nanos: 385000000 }
name	string	Name of the stage	"Race"
description	string	Short description of the stage	"Grosser Preis von Osterreich 2019 Race"
sportId	string	The Sportradar sport ID	"sr:sport: 40"
categoryId	string	The Sportradar category ID	"36"
categoryName	string	The Sportradar category name	"Formula 1"
parentStageName	string	The parent stage name. This will be the name of the Grand Prix.	"Grosser Preis von Osterreich 2019"
stageType	string	The type of the stage. STAGE_TYPE_RACE, STAGE_TYPE_PRACTICE or STAGE_TYPE_QUALIFYING	"STAGE_TYPE_RACE"
infoTypes	repeated Stage InfoType		[]
isInProgress	boolean	Indicates if the stage is in progress and are producing live data. This will be set to true when we process the first event from the stage and set to false after we have processed the last event.	true

GetStageDetails

GetStageDetails should be used when you know the stageId, but you want to know the start time for the stage and the suggested connect time. As a response you will get one Stage message. **It will only return a Stage message in response if you have booked the stage.**

GetStageTimetable

GetStageTimetable returns a list of stages in the given time frame **that you have booked**. You can fetch data from previous stages or follow a live stage.

We recommend that you call GetStageTimetable at least once every day, but no more than once every 5 minutes, with a time range of 24 hours. As a response you will get a list of stages you have booked. Each stage will have a "startEvents" property of type google.protobuf.Timestamp. This indicates the suggested connection time for this stage. At that time we will start sending live data for that stage. You can connect to the stage up to ONE HOUR before the stage start time, however we don't expect any events before the suggested connection time "startEvents". If you try to connect with StreamEvents RPC before this time you will get an error. If you do; implement a 5-10 second back-off and reconnect.

EventStream

The EventStream service has two procedures: StreamEvents and ReplayStreamEvents. StreamEvents are used for live consumption of data, ReplayStreamEvents are used for integration testing. They both return the same stream of EventResponse, however there are a couple of unique features for ReplayEventStream that makes it suitable for integration testing.

Event streams

For this service you establish one connection and stream events for an individual stage. If you want to consume multiple stages at the same time you have to create several connections.

Note that F1 only have one stage live at any given time. There should be no need to connect to several stages at the same time.

ReplayStreamEvents replays a booked and completed stage with events timing mimicking the real event timing of the stage. You can also request previous races from the EventStream rpc, however this will not mimic the real timing of the stage, instead it will stream the data as fast as possible.

Reconnect

For both StreamEvents and ReplayStreamEvents it is crucial that you automatically reconnect if you lose connection while the stage is ongoing.

You need to keep track of the id of the last event you received. When you reconnect you specify that you want events with an id after the last one you received. This ensures that you don't lose any data if you lose the connection.

Stream Timeout

Each streaming connection is allowed to live up to 6 hours. That ensures that idle connections don't affect the availability of the service. In any case you are encouraged to close the stream when you have received the EndOfStageEvent. That events also includes a deadline where we will close the connection from the server side if you have not closed it already.

StreamEvents

This RPC streams all events for a stage as soon as they are available. For a stage that is in progress this will be in real-time. For a completed stage this will be as fast as the network connection allows since all the events are available at the time of the request.

Reconnect

We don't expect you to need to reconnect during a live stage, however you have to be prepared to do so should it be needed. In the unlikely case of server crash or other unexpected issues you will be disconnected. In this case you'll be able to reconnect to another server in milliseconds and you are able to continue from where you lost connection without any data loss.

EventsRequest			
Property	**Type**	**Description**	**Optional /Mandatory**
stageId	string	The stageId for the stage you want to stream events from	Mandatory
eventTypes	repeated string	Event type filter. Leave empty if you want to receive all event types. Add event names to the list if you only want specific types of events. E.g. ["SessionTimeEvent", "StageStatusEvent", "WeatherUpdateEvent"] StartOfStageEvent, EndOfStageEvent, EarlyBetStartEvent, BetStartEvent, and BetStopEvent can not be filtered out. These 5 event types will always be sent no matter what events you specify in the filter.	Mandatory - defaults to []
afterSequenceId	int64	The id of the last event you received (found as "id" in EventWrapper for the event). Set to -1 to receive only live events. Set to 0 to receive all events from the beginning of the stage.	Mandatory - defaults to 0

ReplayStreamEvents

ReplayStreamEvents is tailor made for testing and integration. It mimics a real stage by making sure that the timing of the events match what you would have seen if you where connected to a live stage. All the events sent are taken from real stages, and except from the date this is as close to streaming a live stage as you can get. We are aware that you sometimes want to test things a bit faster than 1:1 speed. For that reason ReplayStreamEvents also have a "fast-forward" parameter, "speedFactor" that allows you to play back the race in 1-10x normal speed. As with StreamEvents you can use the afterSequenceId parameter in the request to start streaming from any given point in the stage. That allows you to stop the replay at any point and start the replay back up again from the same place in the stage when you are ready.

Since ReplayStreamEvents RPC is made for testing we have implemented a random disconnect on the server side. This allows you to properly test your reconnection logic before you move into production. The stream from StreamEvents and ReplayStreamEvents are exactly the same. So once you are ready to move this into production you only have to change the RPC name from ReplayStreamEvents to StreamEvents and use the EventsRequest parameter message instead of the ReplayEventsRequest.

Reconnect

To ensure that you do implement a logic that handles reconnection the ReplayStreamEvents is built in such a way that **you will get disconnected at random intervals**. It's made this way so that you can test that your integration is able to handle situations like this. This RPC should be used for testing and integration only, and for that reason this random disconnect will not affect live stages.

Make sure to read the Errors section to find the gRPC status codes that you should use to trigger your reconnection logic.

ReplayEventsRequest			
Property	**Type**	**Description**	**Optional /Mandatory**
stageId	string	The stageId for the stage you want to stream events from	Mandatory
eventTypes	repeated string	Event type filter. Leave empty if you want to receive all event types. Add event names to the list if you only want specific types of events. E.g. ["SessionTimeEvent", "StageStatusEvent", "WeatherUpdateEvent"]	Mandatory - defaults to []
afterSequenceId	int64	The id of the last event you received (found as "id" in EventWrapper for the event). Set to -1 to receive only live events. Set to 0 to receive all events from the beginning of the stage.	Mandatory - defaults to 0
speedFactor	int32	The speed of the replay. A value of 1 equals normal "real-time" speed for the replay. A value of 5 equals 5x normal speed. Allowed values are 1-10. Any other values will be interpreted as normal speed: 1.	Mandatory - defaults to 0

StreamCarPositionEvents

This RPC streams car position events for a stage in real time. Given the high frequency nature of this stream, we allow for an extra request parameter 'periodMs' which specifies the time delay between consecutive events in the response.

CarPositionEventsRequest

Property	Type	Description	Optional /Mandatory
stageId	string	The stageId for the stage you want to stream events from	Mandatory
periodMs	int32	Specify the time delay between the response messages in milliseconds, e.g. if periodMs = 500, the resolution will be of 2 messages per second. Minimum period is 20 ms, and the response will use a period corrected to the next multiple of 20ms from the provided period (e.g. is periodMs = 50ms, the response will consist of messages every 60ms).	Mandatory - defaults to 0
afterSequenceId	int64	The id of the last event you received (found as "id" in EventWrapper for the event). Set to -1 to receive only live events.	Mandatory - defaults to 0

StageInfo

StageInfo service has one procedure that allows you to get a snapshot of an ongoing race. This is used to get the state of the race quickly, and then connecting to the stream to update the state.

GetStageSnapshot

Pass in a StageSnapshotRequest message as a parameter to the GetStageSnapshot procedure. In response you'll get a StageSnapshotResponse that contains the most important information about the current state of the race. The id of the most recent event used to build the snapshot will be included in the response. Use this sequenceId to start streaming events to continue streaming from the state the stage was in when this snapshot was created.

GetStateSnapshot and Replays

When you request a snapshot from a completed stage or during a replay you will get the current, real, state of the stage. For completed stages this is the state after the stage was FINALISED.

StageSnapshotRequest

Property	Type	Description	Optional/Mandatory
stageId	string	The stageId for the stage you want to stream events from	Mandatory

StageSnapshotResponse			
Property	**Type**	**Description**	**Optional /Mandatory**
raceLeaderboardEvent or qualifyingLeaderboardEvent or practiceLeaderboardEvent	sportradar.ldi.f1.events.v1.RaceLeaderboardEvent sportradar.ldi.f1.events.v1.QualifyingLeaderboardEvent sportradar.ldi.f1.events.v1.PracticeLeaderboardEvent	The current leaderboard for the stage. Leaderboards differ slightly for different stage types. The correct leaderboard for the requested stage is sent.	Mandatory
stageStatusEvent	sportradar.ldi.f1.events.v1.StageStatusEvent	The most recent status of the stage.	Mandatory - defaults to UNKNOWN
trackStatusEvent	sportradar.ldi.f1.events.v1.TrackStatusEvent	The most recent track status.	Mandatory - defaults to UNKNOWN
lapCountEvent	sportradar.ldi.f1.events.v1.LapCountEvent	The most recent lap count.	Mandatory
weatherUpdateEvent	sportradar.ldi.f1.events.v1.WeatherUpdateEvent	The most recent weather status	Mandatory
sequenceId	int64	The sequenceId of the last event used to generate this snapshot	Mandatory
startingPositionEvent	sportradar.ldi.f1.events.v1.StartingPositionEvent	The drivers starting position in this stage.	Mandatory
earlyBetStartEvent or betStartEvent or betStopEvent	sportradar.ldi.f1.services.v1.EarlyBetStartEvent sportradar.ldi.f1.services.v1.BetStartEvent portradar.ldi.f1.services.v1.BetStopEvent	Current bet status	Mandatory
feedQualityEvent	sportradar.ldi.f1.events.v1.FeedQualityEvent	Current feed quality	Mandatory
sessionTimeEvent	sportradar.ldi.f1.events.v1.SessionTimeEvent	Session time	Mandatory

Sample response

Sample snapshot response converted to JSON

```
{
    "raceLeaderboardEvent": {
        "stageId": "<stageid>",
        "isPartialUpdate": false,
        "idealLapTime": "1:16.127",
        "items": [
            {
                "position": 1,
                "driverData": {
                    "driverId": "41600",
                    "racingNumber": 77,
                    "numberOfTyres": 3,
                    "position": 1,
                    "lastLapTime": "1:18.325",
                    "personalBestLapTime": "1:18.272",
                    "personalBestLapNumber": "54",
                    "pitStops": 4,
                    "lapsCompleted": 64,
                    "tyre": "SOFT",
                    "isActive": false
                }
            },
            {
                "position": 2,
                "driverData": {
                    "driverId": "39412",
                    "racingNumber": 27,
                    "numberOfTyres": 4,
                    "position": 2,
                    "lastLapTime": "1:29.853",
                    "personalBestLapTime": "1:29.576",
                    "personalBestLapNumber": "18",
                    "pitStops": 3,
                    "lapsCompleted": 64,
                    "tyre": "HARD",
                    "isActive": false
                }
            },
           // + more drivers
        ]
    },
    "stageStatusEvent": {
        "state": 5
    },
    "trackStatusEvent": {
        "trackStatus": 1,
        "message": "AllClear"
    },
    "lapCountEvent": {
        "totalracelaps": 64,
        "currentracelap": 64,
        "racelapsremaining": 0
    },
    "weatherUpdateEvent": {
        "humidity": 86.5,
        "rainfall": false,
        "airTemp": 21.9,
        "pressure": 992.3,
        "trackTemp": 26.8,
        "windDirection": 347,
        "windSpeed": 0.7
    },
    "sequenceId": 684439,
    "startingPosition": {
        "items": [
            {
                "position": 1,
                "driverData": {
                    "driverId": "41600",
                    "racingNumber": 77,
                    "numberOfTyres": 0,
                    "position": 1,
                    "lastLapTime": "",
                    "personalBestLapTime": "",
                    "personalBestLapNumber": "",
                    "pitStops": 0,
                    "lapsCompleted": 0,
                    "tyre": "WET",
                    "isActive": false
                }
            },
            {
                "position": 2,
                "driverData": {
                    "driverId": "39412",
                    "racingNumber": 27,
                    "numberOfTyres": 0,
                    "position": 2,
                    "lastLapTime": "",
                    "personalBestLapTime": "",
                    "personalBestLapNumber": "",
                    "pitStops": 0,
                    "lapsCompleted": 0,
                    "tyre": "WET",
                    "isActive": false
                }
            },
           // + more drivers
        ]
    },
        "betStartEvent":{
                reason: ""
        }
}
```

GetStageTimelineEvents

Return all Timeline events for the requested stage in the requested timeframe. These event are not too frequent so it is possible to request all event types for a full race.

Allowed Timeline events types are:
StageStatusEvent, TrackStatusEvent, RaceControlEvent, LapCountEvent, FastestLapAchievedEvent, FastestSpeedAchievedEvent,
PitLaneTimeEvent, FastestSectorTimeAchievedEvent, DriverOutEvent, DriverPitStopEvent, DriverStoppedEvent,
Top3DriversEvent, OvertakeEvent, StartedRainingEvent

GetStageTimelineEventsRequest			
Property	**Type**	**Description**	**Optional /Mandatory**
stageId	string	The stageId for the stage you want to stream events from	Mandatory
eventTypes	repeated string		Mandatory
from	google.protobuf. Timestamp		Mandatory
to	google.protobuf. Timestamp		Mandatory

GetStageCarPositionEvents

Returns CarPositionEvents within the provided time range and with the provided delay between timestamps.

The maximum allowed length of the provided time range is 1 minute (i.e. from + 1 minute > to).

GetStageCarPositionEventsRequest			
Property	**Type**	**Description**	**Optional /Mandatory**
stageId	string	The stageId for the stage you want to stream events from	Mandatory
from	google. protobuf. Timestamp		Mandatory
to	google. protobuf. Timestamp		Mandatory
periodMs	int32	Specify the time delay between the response messages in milliseconds, e.g. if periodMs = 500, the resolution will be of 2 messages per second. Minimum period is 20 ms, and the response will use a period corrected to the next multiple of 20ms from the provided period (e.g. is periodMs = 50ms, the response will consist of messages every 60ms).	Mandatory

GetTrackModelURLForStage

Returns a presigned S3 URL to download the csv track model for the stage. The csv model contains the ENU for the points of the track, together with a 'LAYER' property indicating the part of the track. The usage of this model is explained in DriverCarPosition section in the event reference.

`TrackModelRequest`			
Property	**Type**	**Description**	**Optional/Mandatory**
stageId	string	The stageId for the requested track	Mandatory

Errors

Errors for the service follows the gRPC standard for status codes. https://github.com/grpc/grpc/blob/master/doc/statuscodes.md

To the right in this table you can see a column named "Trigger reconnect". This indicates whether you should trigger a reconnect or not if you get this error message.

Code	Number	Description	Sample situation	Trigger reconnect
OK	0	Not an error. Successful request.		NO
CANCELLED	1	The operation was cancelled		NO
UNKNOWN	2	Unknown error.		Yes, 10-100ms back-off
INVALID_ARGUMENT	3	Malformed or invalid argument (s).	When you try to call an RPC with invalid arguments	NO
DEADLINE_EXCEEDED	4	The deadline expired before the operation could complete.	When you set a deadline for the response from client side and you don't get a response within the deadline you set.	Yes, 10-100ms back-off
NOT_FOUND	5	The requested resource could not be found.	When you try to call StreamEvents or ReplayStreamEvents for a stage that has not started yet (data does not exist)	If it's an upcoming stage, back off for 10s and reconnect
ALREADY_EXISTS	6		Not used	NO
PERMISSION_DENIED	7	The authentication credentials used for this operation is not authorized to perform the operation.	When you try to request data from a stage that you have not booked.	NO
UNAUTHENTICATED	16	The request does not have valid authentication credentials for this operation	When you try to call an RPC with missing or invalid SSO token.	NO
RESOURCE_EXHAUSTED	8	You have exceeded your quota for concurrent stream or requests		NO
FAILED_PRECONDITION	9	The operation was rejected because the system is not in a state required for the operation's execution.	When you try to request a snapshot from a stage that has not started yet.	If it's an upcoming stage, back off for 10s and reconnect
ABORTED	10	The operation was aborted, typically due to sequence check failure or transaction abort.		NO
OUT_OF_RANGE	11	The operation attempted was out of range	When you use an afterSequenceId that is greater than the max sequenceId for the requested stage	NO
UNIMPLEMENTED	12	Operation is not implemented		NO
INTERNAL	13	Serious internal error		NO
UNAVAILABLE	14	Service unavailable		YES, with a back-off of 1s or more
DATA_LOSS	15	Unrecoverable data loss or corruption	Data-loss over network or if you have changed the protofiles	YES. with a back-off of 1s or more

Rate Limiting

We have a rate limiting server setup per service with the following refill rates:

Service	Refill rate
StageInfo	50/sec
StageDiscovery	25/sec
EventStream	10/sec

All have a burst factor of 4, meaning that a client can temporally request up to 4 time the refill rate. Blocked clients will receive an error code: UNAVAILABLE if rate limited.

Events

A reference document for the F1 events is found here: F1 Event Reference

Code samples

The following samples show how to consume from the service API in the Java programming language. For examples in other languages, as well as in-depth info, please refer to https://grpc.io/docs/.

Preliminiaries

In the sections below we show how to build your integration code using Maven. Knowledge of Maven is assumed.

Compiling protos

After you have acquired the `.proto` files from Sportradar, put them in `src/main/proto/` in your project and include the following or similar in the `<dependencies>` and `<build>` sections of your `pom.xml` file. This will build Java artifacts for the protos and gRPC layer and add to `target/generated-sources/`.

```
...

<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
  <version>1.3.2</version>
</dependency>
<dependency>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.6.1</version>
</dependency>

...

<build>
...

<extensions>
  <extension>
    <artifactId>os-maven-plugin</artifactId>
    <groupId>kr.motd.maven</groupId>
    <version>1.6.2</version>
  </extension>
</extensions>
...

<plugins>
<plugin>
  <artifactId>protobuf-maven-plugin</artifactId>
  <configuration>
    <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.26.0:exe:${os.
detected.classifier}
    </pluginArtifact>
    <pluginId>grpc-java</pluginId>
    <protocArtifact>com.google.protobuf:protoc:3.11.2:exe:${os.
detected.classifier}
    </protocArtifact>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>compile-custom</goal>
      </goals>
    </execution>
  </executions>
  <groupId>org.xolstice.maven.plugins</groupId>
  <version>0.6.1</version>
</plugin>


...
```

Importing gRPC

In order to import the necessary gRPC libraries, include the following dependency or similar in your `pom.xml` file.

```
<
dependency
>
  <groupId
>io.grpc</
groupId>
  <
artifactId
>grpc-all
</
artifactId
>
  <version
>1.27.1</
version>
</
dependency
>
```

Consuming events using a blocking call

The sample below shows how to make a blocking (in-thread) gRPC call to the F1 service.

```java
package com.sportradar.livedata.integration.f1.service.snippets;

import static com.google.common.net.HttpHeaders.AUTHORIZATION;

import com.google.protobuf.Any;
import com.google.protobuf.InvalidProtocolBufferException;
import com.sportradar.livedata.integration.f1.services.v1.
EventStreamGrpc;
import com.sportradar.livedata.integration.f1.services.v1.ServiceProtos;
import io.grpc.*;
import io.grpc.stub.MetadataUtils;

/**
 * Example of consuming events from Sportradar F1 service using a
blocking (synchronous) gRPC call.
 */
public class DocSnippetBlockingCall {

  private static final String AUTH_TOKEN = "token-from-sportradar-sso";

  /** Demonstrate streaming events from Sportradar F1 service using a
blocking gRPC call. */
  public void streamEventsGrpcBlocking() {
    // Set up a network channel
    ManagedChannel channel =
        ManagedChannelBuilder.forAddress("stream.ld.betradar.com", 443)
            .build();

    // Make metadata object containing authorization header
    Metadata headers = new Metadata();
    headers.put(Metadata.Key.of(AUTHORIZATION, Metadata.
ASCII_STRING_MARSHALLER), AUTH_TOKEN);

    // Creates the client stub (proxy for network service)
    EventStreamGrpc.EventStreamBlockingStub stub =
        MetadataUtils.attachHeaders(EventStreamGrpc.newBlockingStub
(channel), headers);

    // Make the call and output resulting events continuously
    ServiceProtos.EventsRequest request =
        ServiceProtos.EventsRequest.newBuilder().setStageId("test:stage:
6538").build();
    stub.streamEvents(request)
        .forEachRemaining(
            response -> {
              System.out.println(
                  "Got event of type "
                      + response.getEventWrapper().getEventType()
                      + " with id "
                      + response.getEventWrapper().getId());

              // Unpack BetStartEvent proto (as example) and print
reason
              if (response
                  .getEventWrapper()
                  .getEvent()
                  .getTypeUrl()
                  .equals("type.googleapis.com/sportradar.ldi.f1.
services.v1.BetStartEvent")) {
                try {
                  ServiceProtos.BetStartEvent betStart =
                      response
                          .getEventWrapper()
                          .getEvent()
                          .unpack(ServiceProtos.BetStartEvent.class);
                  System.out.println("Got bet start with reason: " +
betStart.getReason());
                } catch (InvalidProtocolBufferException e) {
                  e.printStackTrace();
                }
              }
            });
  }

  public static void main(String[] args) {
    new DocSnippetBlockingCall().streamEventsGrpcBlocking();
  }
}
```

Consuming events using a non-blocking call

The sample below shows how to make a non-blocking gRPC call to the F1 service.

```
package com.sportradar.livedata.integration.f1.service.snippets;

import static com.google.common.net.HttpHeaders.AUTHORIZATION;

import com.google.protobuf.InvalidProtocolBufferException;
import com.sportradar.livedata.integration.f1.services.v1.
EventStreamGrpc;
import com.sportradar.livedata.integration.f1.services.v1.ServiceProtos;
import io.grpc.*;
import io.grpc.stub.MetadataUtils;
import io.grpc.stub.StreamObserver;

/**
 * Example of consuming events from Sportradar F1 service using a non-
blocking (asynchronous) gRPC
 * call.
 */
public class DocSnippetNonblockingCall {

  private static final String AUTH_TOKEN = "token-from-sportradar-sso";

  /** Demonstrate streaming events from Sportradar F1 service using a
non-blocking gRPC call. */
  public void streamEventsGrpcNonblocking() throws InterruptedException
{
    // Set up a network channel
    ManagedChannel channel =
        ManagedChannelBuilder.forAddress("stream.ld.betradar.com", 443)
            .build();

    // Make metadata object containing authorization header
    Metadata headers = new Metadata();
    headers.put(Metadata.Key.of(AUTHORIZATION, Metadata.
ASCII_STRING_MARSHALLER), AUTH_TOKEN);

    // Creates the client stub (proxy for network service)
    EventStreamGrpc.EventStreamStub stub =
        MetadataUtils.attachHeaders(EventStreamGrpc.newStub(channel),
headers);

    // Make the call and output resulting events continuously
    ServiceProtos.EventsRequest request =
        ServiceProtos.EventsRequest.newBuilder().setStageId("test:stage:
6538").build();

    // Make the observer of responses
    StreamObserver<ServiceProtos.EventResponse> observer = new
EventObserver();

    // Call service
    stub.streamEvents(request, observer);

    // Sleep 10 secs while the observer handles some events
    Thread.sleep(10000);
  }

  public static void main(String[] args) throws InterruptedException {
    new DocSnippetNonblockingCall().streamEventsGrpcNonblocking();
  }

  private class EventObserver implements StreamObserver<ServiceProtos.
EventResponse> {
    @Override
    public void onNext(ServiceProtos.EventResponse response) {
      System.out.println(
          "Got event of type "
              + response.getEventWrapper().getEventType()
              + " with id "
              + response.getEventWrapper().getId());

      // Unpack BetStartEvent proto (as example) and print reason
      if (response
          .getEventWrapper()
          .getEvent()
          .getTypeUrl()
          .equals("type.googleapis.com/sportradar.ldi.f1.services.v1.
BetStartEvent")) {
        try {
          ServiceProtos.BetStartEvent betStart =
              response.getEventWrapper().getEvent().unpack
(ServiceProtos.BetStartEvent.class);
          System.out.println("Got bet start with reason: " + betStart.
getReason());
        } catch (InvalidProtocolBufferException e) {
          e.printStackTrace();
        }
      }
    }

    @Override
    public void onError(Throwable throwable) {
      System.out.println("Got error: " + throwable);
    }

    @Override
    public void onCompleted() {
      System.out.println("Done");
    }
  }
}
```

Back to top