MG - Unified odds SDK (Software developer kit)

The Unified Odds SDK (Software developer kit) provides a simple and efficient way for bookmakers to access Betradar's odds and sport information. It combines subscription of messages and API calls into a unified Java or .NET interface that hides most of the complexity, including recovery.

Table of contents

Info

Our SDK has a dedicated documentation section both for the Java and .NET versions. For more detailed information about using and getting started with the SDK, please visit our dedication SDK sections:

- http://sdk.sportradar.com/unifiedfeed/java2
- http://sdk.sportradar.com/unifiedfeed/net

Main SDK benefits over protocol/API

- The SDK hides the separation between messages and API-lookup. The client system just receives the message objects where all information is filled in by the SDK caches, and in some cases looked up in the background when your system requests some more rarely requested information.
- The SDK takes care of translations transparently. The client system just needs to define what languages it needs.
- The SDK takes care of dynamic text markets and outright markets automatically, which requires some extra logic and lookup for someone not using the SDK.
- The SDK handles initial connect and state, as well as recovery in case of a temporary disconnect. This needs to be handled manually by someone not using the SDK.
- The SDK provides an up to date cache of each sport-events current status that is updated automatically.

Getting started with the SDK

In order to get started quickly with the SDK, we suggest using the example projects which can be obtained from our SDK websites (please check above).
The example projects already contain the basic logic for listeners/events implementation and already suggest how to integrate logging frameworks with it. Therefore, only a valid UOF token is needed to do a quick start and get first messages from the feed or the API.

Main components of the SDK

The following components are mainly used in the SDK. Please note that names of classes and interfaces can slightly differ between Java and .NET, but mostly do provide the same functionalities.

Configuration

Before the SDK can connect to the feed and starts processing messages it has to be properly configured. The configuration must contain various information from the access token used to authenticate with the feed to various user preferences like default language, supported languages, ... The SDK supports both in-code and (external) file-based configuration. The configuration uses a builder pattern forcing the user to set mandatory values while allowing to skip optional ones. Various builders are used to create different configurations which target different environments:

- Staging environment: Configuration for staging environment is build using the IConfigurationBuilder (.Net) or ConfigurationBuilder (Java)
- Production environment: the same as above
- Replay server: Configuration for the replay server is build using the IReplayConfigurationBuilder or ReplayConfigurationBuilder (Java)
- Non sportradar environment: Sometimes (especially for testing) can be useful to connect to a locally hosted server. Configuration for custom environment can be build using ICustomConfigurationBuilder or CustomConfigurationBuilder (Java)

Sessions

A session represents a channel of communication with the AMQP broker. Each session runs in its own thread meaning that messages received by different sessions are processed in different threads. The threads are owned by the component used to communicate with the feed, meaning that long-running operations should not be invoked on those threads because this blocks receiving new messages on that session. More than one session can be created on each feed instance. The session itself cannot be opened or closed (started or stopped) it opens/closes when the associated feed object is opened/closed. When building a session, a message interest specifying the session scope must be provided. Message interest specifies which messages will be received by the associated session. Only certain combinations of message interest/session combinations are permitted

- Any one session
- High priority + Low priority
- Any combination of Live messages, Prematch messages and Virtual sports

Recovery

The recovery operation ensures that the state of the SDK (and therefore the system using the SDK) is in-sync with the state of the feed. Whether a given producer is in-sync with the feed can be determined by invoking an appropriate method on the producer object. Each time the state of the producer changes, the user is notified via global event / callback. When the SDK starts, the recovery needs to be initiated in order to get the current state. Since the SDK does not persist any information, a user must provide a timestamp from which the recovery will be made (a setProducerRecoveryFromTimestamp method in Java & AddTimestampBeforeDisconnect method in .Net).

No additional action from the user is required to ensure the state of the SDK is in-sync with the state of the feed. Once the recovery is made and the producer is "marked as up", the "recovery timestamp" can be determined by calling an appropriate method on the producer object(getTimestampForRecovery in Java & LastTimestampBeforeDisconnect in .Net). This value should be periodically retrieved and stored to ensure it is available and up-to-date in case a restart of the system is required. The recovery operations have a time limit - indicating the maximum time window in which they need to complete. If the recovery is not completed in a specified time window, it is automatically restarted by the SDK. The maximum recovery time can be configured via SDK configuration**.**

Logging

The SDK logging component is configured via configuration file for the selected logging platform. The examples available on the http://sdk.sportradar.com use logback(Java version) and log4net(.net version) and can be used as reference for configuration of those two logging platforms. The actual logging platform must be provided by the user via dependency management system. Since .Net does not provide a logging facade (equivalent of SLF4J in java) an open-source library Common.Logging is used. Since investigation of possible SDK issues relies on the logs generated by the SDK, the users are encouraged to keep the logs for longer periods and have them configured to log as much information as possible (especially during the integration). SDK logs data to multiple files and different levels can be assigned to different loggers.

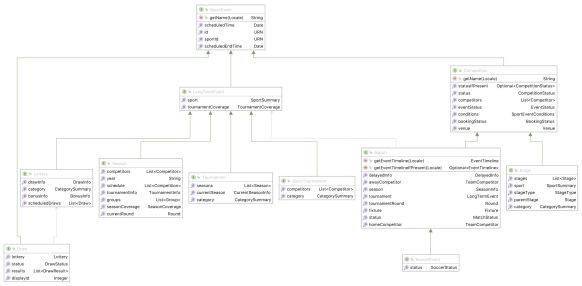
- Execution log – contains log entries providing information of the SDK flow for a given moment
- Interaction log – logs the user interaction with the SDK. This includes when a key method on a SDKs public interface is invoked or when a message is dispatched to the user.
- Rest traffic: logs all requests to the Sports API and the associated responses. This can produce a lot of log entries so the users are encouraged to decrease the level once in production
- Feed traffic: logs all messages received from the AMQP broker. This can produce a lot of log entries so the users are encouraged to decrease the level once in production
- Traffic failure: Logs the messages which could not be dispatched to the user.

Error handling

SDK exception handling can be configured via the SDK configuration. By default, any exception thrown during the exection of a publicly available method is caught by the SDK and a null reference is returned to the user. When the config option is set to "throw", all exceptions are propagated to the caller. For more information see the *ExceptionHandlingStrategy* enum. The exception to this rule are methods on the feed object which are called before the feed is opened - most notably the open metho d in java and Open in .Net. Those two methods always throw an exception if the connection to the feed cannot be established.

Hierarchy of types representing sport events

The SDK exposes a number of types used to represent different types of sport events(matches, seasons, competitions,...). When event specific message dispatchment is not used, the the sport event is represented by it's base type and needs to be type-casted into a more specific type. The class diagram below, shows all interfaces derived from the sport event along with methods defined on each one.



Back to top