Идентификация торговых инструментов

Для точного определения различных торговых инструментов используются различные идентификаторы:

**Isin**(*англ.:* International Security Identification Number) — международный идентификационный номер ценной бумаги, состоит из 12 символов цифр и латинских букв, которые начинаются, как правило, с 2-буквенного кода страны эмитента ценной бумаги.  
**Структура ISIN:**  
Первые два символа — буквы, определяющие код страны эмитента согласно стандарту ISO 3166-1 (например, российские ISIN-коды начинаются буквами RU).  
Далее следуют 9 букв и цифр национального идентификационного кода ценной бумаги (англ. National Securities Identifying Number, NSIN). Завершает код контрольная цифра.  
Например, ISIN акции TCS Group — US87238U2033.

**Ticker** — краткое (1-5 букв) наименование ценной бумаги *на конкретной бирже*. Без указания биржи и режима торгов, по сути, является бессмысленным набором букв. Для этого на российских биржах MOEX и SPBE используется специальный признак "Режим торгов", который соответствует техническому термину "class\_сode". Комбинация тикера и "class\_code" является уникальным идентификатором. Отдельно же, только тикер, - нет.  
Например, тикер TCS Group Tinkoff - TCSG.

**FIGI**(*англ.:* Financial Instrument Global Identifier) — глобальный идентификатор финансового инструмента. Представляет собой 12-символьный код из латинских букв и цифр, определяется как идентификатор ценной бумаги на торговой площадке (бирже), которая является некоторым *"источником цен"*.

Важно: не всегда FIGI, присваемые в Тинькофф Инвестициях, совпадают с международным классификатором. Является устаревшим параметром и не рекомендуется к использованию.

Идентификаторы из модели финансовых инструментов Тинькофф Инвестиций

В Тинькофф Инвестициях применяется иерархическая система идентификации финансовых объектов. На верхнем уровне находится **актив** - это по сути работающая на финансовом рынке компания, являющаяся эмитентом ценных бумаг. Примером актива может быть например Сбербанк.

У актива может несколько финансовых инструментов: акции (в т.ч. на разных биржах), облигации и т.д., каждая из подобных ценных бумаг называется **позиция**. То есть *Позиция* - это совокупность актива, типа и торговой площадки(биржи).

Но одна и та же позиция может торговаться в разных режимах торгов и по разной цене: например: валюта разной лотности может иметь различную стоимость; или акцию можно продать за биржевом рынке, так и на внебиржевом (в торговле на выходных). Поэтому есть еще один идентификатор - **торговый инструмент** - он определяется как ценная бумага на торговой площадке (бирже) в указанном режиме торгов. Получить котировки можно только по торговому инструменту.

Подведем итоги,

* идентификатор актива - это asset\_uid
* идентификатор позиции - это position\_uid
* идентификатор торгового инструмента - это instrument\_uid

Для удобства клиентов, использовавших в своих системах устаревший идентификатор FIGI в многих методах API Тинькофф предусмотрено универсальное поле **Instrument\_Id** - в которое можно передавать как FIGI, так и instrument\_uid.

Вы можете получить идентификаторы конкретного инструмента с помощью метода FindInstrument, передав в параметр *query* известный идентификатор или название инструмента.

Использование Instrument\_Id является более предпочтительным, так как предполагает возможность использования **instrument\_uid** или **FIGI**.

На данный момент при передаче значения в параметр Instrument\_Id сначала выполняется поиск инструмента по **instrument\_uid**, а потом уже по **FIGI**.

**Основным идентификатором торгового инструмента при работе с TINKOFF INVEST API является *UID*, так как, например, в опционах идентификатор FIGI не поддерживае.**

**\_\_\_\_\_\_\_**

**Учебное пособие по основам**

Базовое учебное пособие по gRPC-web.

В этом руководстве представлено базовое описание использования gRPC-Web в браузерах.

Ознакомившись с этим примером, вы узнаете, как:

* Определите службу в файле .proto.
* Сгенерируйте клиентский код, используя компилятор буфера протокола.
* Используйте gRPC-Web API, чтобы написать простой клиент для вашего сервиса.

Это предполагает мимолетное знакомство с буферами протокола.

Зачем использовать gRPC и gRPC-Web?

С помощью gRPC вы можете один раз определить свой сервис в файле .proto и реализовать клиентов и серверы на любом из поддерживаемых gRPC языков, которые, в свою очередь, могут работать в самых разных средах — от серверов в крупном центре обработки данных до вашего собственного планшета. Все сложности взаимодействия между разными языками и средами берёт на себя gRPC. Вы также получаете все преимущества работы с буферами протокола, включая эффективную сериализацию, простой IDL и лёгкое обновление интерфейса. gRPC-Web позволяет получать доступ к сервисам gRPC, созданным таким образом, из браузеров с помощью стандартного API.

Определите услугу

Первым шагом при создании сервиса gRPC является определение методов сервиса и типов сообщений запросов и ответов с помощью буферов протокола. В этом примере мы определяем EchoService в файле с именем echo.proto. Дополнительную информацию о буферах протокола и синтаксисе proto3 можно найти в документации по protobuf.

**message** **EchoRequest** {

**string** **message** = 1;

}

**message** **EchoResponse** {

**string** **message** = 1;

}

**service** EchoService {

**rpc** Echo(EchoRequest) **returns** (EchoResponse);

}

Реализовать Внутренний сервер gRPC

Далее мы реализуем наш интерфейс EchoService с помощью Node в серверной части gRPC EchoServer. Это позволит обрабатывать запросы от клиентов. Подробности см. в файле node-server/server.js.

Вы можете реализовать сервер на любом языке, поддерживаемом gRPC. Пожалуйста, ознакомьтесь с главной страницей для получения более подробной информации.

**function** doEcho(call, callback) {

callback(**null**, {message: call.request.message});

}

Настройте прокси-сервер Envoy

В этом примере мы будем использовать прокси-сервер Envoy для переадресации запроса браузера gRPC на серверную часть. Полный файл конфигурации можно посмотреть в envoy.yaml

Чтобы пересылать запросы gRPC на сервер, нам нужен такой блок:

**admin**:

**address**:

**socket\_address**: { **address: 0.0.0.0, port\_value**: 9901 }

**static\_resources**:

**listeners**:

- **name**: listener\_0

**address**:

**socket\_address**: { **address: 0.0.0.0, port\_value**: 8080 }

**filter\_chains**:

- **filters**:

- **name**: envoy.http\_connection\_manager

**typed\_config**:

**"@type":** type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager

**codec\_type**: auto

**stat\_prefix**: ingress\_http

**route\_config**:

**name**: local\_route

**virtual\_hosts**:

- **name**: local\_service

**domains**: ["\*"]

**routes**:

- **match**: { **prefix**: "/" }

**route**: { **cluster**: echo\_service }

**http\_filters**:

- **name**: envoy.grpc\_web

**typed\_config**:

**"@type":** type.googleapis.com/envoy.extensions.filters.http.grpc\_web.v3.GrpcWeb

- **name**: envoy.filters.http.router

**typed\_config**:

**"@type":** type.googleapis.com/envoy.extensions.filters.http.router.v3.Router

**clusters**:

- **name**: echo\_service

**connect\_timeout**: 0.25s

**type**: LOGICAL\_DNS

**typed\_extension\_protocol\_options**:

**envoy.extensions.upstreams.http.v3.HttpProtocolOptions**:

**"@type":** type.googleapis.com/envoy.extensions.upstreams.http.v3.HttpProtocolOptions

**explicit\_http\_config**:

**http2\_protocol\_options**: {}

**lb\_policy**: ROUND\_ROBIN

**load\_assignment**:

**cluster\_name**: echo\_service

**endpoints**:

- **lb\_endpoints**:

- **endpoint**:

**address**:

**socket\_address**:

**address**: node-server

**port\_value**: 9090

Возможно, вам также потребуется настроить CORS, чтобы браузер мог запрашивать контент из разных источников.

В этом простом примере браузер отправляет запросы gRPC на порт :8080. Envoy перенаправляет запрос на сервер gRPC, прослушивающий порт :9090.

Генерируйте сообщения Protobuf и заглушку клиента Обслуживания

Чтобы сгенерировать классы сообщений protobuf из нашего echo.proto, выполните следующую команду:

protoc -I=$DIR echo.proto **\**

--js\_out=import\_style=commonjs:$OUT\_DIR

Опция import\_style, переданная флагу --js\_out, гарантирует, что в сгенерированных файлах будут присутствовать операторы require() в стиле CommonJS.

Чтобы сгенерировать заглушку клиента gRPC-Web, сначала вам понадобится плагин gRPC-Web protoc. Чтобы скомпилировать плагин protoc-gen-grpc-web, вам нужно запустить его из корневого каталога репозитория:

cd grpc-web

sudo make install-plugin

Чтобы сгенерировать файл-заглушку клиента службы, выполните эту команду:

protoc -I=$DIR echo.proto **\**

--grpc-web\_out=import\_style=commonjs,mode=grpcwebtext:$OUT\_DIR

В --grpc-web\_out приведенном выше параметре:

* mode может быть grpcwebtext (по умолчанию) или grpcweb
* import\_style может быть closure (по умолчанию) или commonjs

Наша команда по умолчанию генерирует заглушку клиента для файла echo\_grpc\_web\_pb.js.

Написать клиентский код JS

Теперь вы готовы написать клиентский код на JS. Поместите его в client.js файл.

**const** {EchoRequest, EchoResponse} = require('./echo\_pb.js');

**const** {EchoServiceClient} = require('./echo\_grpc\_web\_pb.js');

**var** echoService = **new** EchoServiceClient('http://localhost:8080');

**var** request = **new** EchoRequest();

request.setMessage('Hello World!');

echoService.echo(request, {}, **function**(err, response) {

*// ...*

});

Вам понадобится package.json файл

{

**"name"**: "grpc-web-commonjs-example",

**"dependencies"**: {

**"google-protobuf"**: "^3.6.1",

**"grpc-web"**: "^0.4.0"

},

**"devDependencies"**: {

**"browserify"**: "^16.2.2",

**"webpack"**: "^4.16.5",

**"webpack-cli"**: "^3.1.0"

}

}

Скомпилируйте библиотеку JS

Наконец, собрав всё это воедино, мы можем скомпилировать все необходимые JS-файлы в одну библиотеку JS, которую можно использовать в браузере.

npm install

npx webpack client.js

Теперь встроите dist/main.js в свой проект и посмотрите на него в действии!

Последнее изменение 25 ноября 2024 года: функция: перенос индикатора строки $ в scss (#1354) (ab8b3af)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_----**