Building Trading Bots Using Java

Shekhar Varshney

Apress®

Building Trading Bots Using Java

Shekhar Varshney

Apress®

Building Trading Bots Using Java

Shekhar Varshney
Granges
Switzerland

ISBN-13 (pbk): 978-1-4842-2519-6 ISBN-13 (electronic): 978-1-4842-2520-2
DOI 10.1007/978-1-4842-2520-2

Library of Congress Control Number: 2016961228

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Kezia Endsley
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Dedicated to the angels in my life:
my mother, my wife Preshita, and my two daughters Mihika and Anya.
Last but not the least, my college professor, Dr. Rajat Moona,
who sowed the seeds of computer programming in my DNA.

Contents at a Glance

Contents

About the Author

Shekhar Varshney is a freelance software developer based in Switzerland with over 19 years of development experience. He started his journey with IBM mainframes, correcting COBOL programs infected with the Y2K bug. At present, his main software development focus is building enterprise services based on SOA principles.

He has a keen interest in software design and architecture. In his free time, he loves experimenting with new APIs and frameworks mostly in the Java ecosystem.

CHAPTER 1

Introduction to Trading Bot

Welcome to the world of automated trading! The fact that you are reading this book suggests that you want to probably build your own bot, which hopefully can make you some money while you are busy with your day job or, like me, want to experiment with the technology that goes into building such a bot using Java.

Automated trading has been around for a while, although it has largely been a preserve of big players such as banks and hedge funds.

This has changed in the last few years, however. Many retail investors are now able to trade on various platforms and exchanges directly, instead of using the services of a traditional broker on the phone, which means the demand has been growing to automate the task of placing orders while these investors get on with their day jobs. As a first step in automating this process, many platforms such as OANDA, LMAX, etc., provide APIs for various programming languages such as Java, Python, C#, and PHP so that the mundane tasks of watching the market, looking at charts, and doing analysis can be automated.

On this journey, we will focus not only on the concepts of automated trading, but also on writing clean, test-driven Java programs.

Toward the end, we will not only have a working trading bot that is ready to trade with any strategy, but from a technical perspective, we will have also gained an appreciation of the event-driven, multithreaded world of Java programming.

Warning Trading foreign exchange on the margin carries a high level of risk, and may not be suitable for all investors. Past performance is not indicative of future results. The high degree of leverage can work against you as well as for you. Before deciding to invest in foreign exchange, you should carefully consider your investment objectives, level of experience, and risk appetite. The possibility exists that you could sustain a loss of some or all of your initial investment and therefore you should not invest money that you cannot afford to lose. You should be aware of all the risks associated with foreign exchange trading and seek advice from an independent financial advisor if you have any doubts.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2520-2_1) contains supplementary material, which is available to authorized users.

S. Varshney, *Building Trading Bots Using Java*, DOI 10.1007/978-1-4842-2520-2_1

What Is a Trading Bot?

In very simple language, a *trading bot* is a computer program that can automatically place orders to a market or exchange, without the need for human intervention. The simplest of bots could be a curl[1] `POST` to an OANDA REST API, such as

```
1   $curl -X POST -d "instrument=EUR_USD&units=2&side=sell&type=market"
    "https://api-fxtrade.oanda.com/v1/accounts/12345/ord\
2   ers"
```

which can be set up on a UNIX cron[2] to run every hour, during trading hours. It has no strategy, nor any external interface or dependencies. It is a one-liner to place an order, which has an equal probability to be in profit or in loss.

On the other end of the spectrum, it could be a complex program based on a distributed architecture, consuming lots of feeds from various sources, analyzing them in realtime and then placing an order. It will be highly available with extremely low latency.

The scale and scope of the bot, as we can see, is varied. To be effective, the bot should be able to accomplish the following tasks:

- Consume market data and/or external news events and social media feeds and distribute them to interested components within the system.
- Have at least one strategy that provides a trading signal.
- Based on a trading signal, place orders with the brokerage platform.
- Account management, i.e., have the ability to keep track of margin requirements, leverage, PNL, amount remaining, etc., in order to curb trading if the amount available breaches a given threshold.
- Position management, i.e., keep track of all currently active positions of various instruments, units of such positions, average price, etc.
- Have the ability to handle events which are triggered by the brokerage platform such as `ORDER_FILLED`, `STOP_LOSS`, etc., and if required take appropriate decisions for such events.
- Some basic monitoring and alerting.
- Some basic risk management. For example, loss limitation by using stop losses for orders or making sure that risk is distributed between risky and safe haven instruments. These are just examples and by no means a comprehensive list of fully managing the risk.

[1] https://en.wikipedia.org/wiki/CURL
[2] https://en.wikipedia.org/wiki/Cron

Why Do We Need a Trading Bot?

I believe most of services provided by exchanges/platforms revolve around the following:

- Market data subscription for instruments of choice and dissemination
- Place orders and trades
- Account and position management
- Historic market data
- Heartbeating
- Callbacks for trade, order, and account events
- Authentication

The trading bot is an attempt to generalize these tasks in a framework and provide an ability to provide the broker/exchange platform specific implementation at runtime, using a dependency injection[3] framework like Spring. Therefore, theoretically speaking, it will just be a change in the Spring configuration file, where we define our implementations for various interfaces that implement these services, and voila, we should be able to support various broker/exchange platforms.

The Capabilities of the Trading Bot

Our bot will have the following capabilities, which are discussed in detail in later chapters:

- Account management
- Integrate with realtime market data feed
- Disseminate of market data
- Place orders
- Handle order/trade and account events
- Analyze of historic prices
- Integrate with Twitter
- Develop strategies

[3] https://en.wikipedia.org/wiki/Dependency_injection

Design Goals

One of the key design goals, alluded to in the beginning of this chapter, is to have the ability to change the implementation of a broker/exchange platform at runtime through Spring configuration. This is possible if we can create specifications for these platform API calls, very similar to the JDBC specification. For example, a sample specification/interface defining the position management requirements are as follows:

```
1  /**
2   * A provider of services for instrument positions. A position for an
      instrument
3   * is by definition aggregated trades for the instrument with an
      average price
4   * where all trades must all be a LONG or a SHORT. It is a useful
      service to
5   * project a summary of a given instrument and also if required close
      all trades
6   * for a given instrument, ideally using a single call.
7   *
8   * The implementation might choose to maintain an internal cache of
      positions in
9   * order to reduce latency. If this is the case then it must find means
      to
10  * either 1) hook into the event streaming and refresh the cache based
      on an
11  * order/trade event or 2) regularly refresh the cache after a given
      time
12  * period.
13  *
14  * @param <M>
15  *      The type of instrumentId in class TradeableInstrument
16  * @param <N>
17  *      the type of accountId
18  *
19  * @see TradeableInstrument
20  */
21 public interface PositionManagementProvider<M, N> {
22
23         /**
24          *
25          * @param accountId
26          * @param instrument
27          * @return Position<M> for a given instrument and accountId(may
              be null if
28          *         all trades under a single account).
29          */
```

```
30          Position<M> getPositionForInstrument(N accountId,
            TradeableInstrument<M> instrument);
31
32          /**
33           *
34           * @param accountId
35           * @return Collection of Position<M> objects for a given
               accountId.
36           */
37          Collection<Position<M>> getPositionsForAccount(N accountId);
38
39          /**
40           * close the position for a given instrument and accountId.
               This is one shot
41           * way to close all trades for a given instrument in an
               account.
42           *
43           * @param accountId
44           * @param instrument
45           * @return if the operation was successful
46           */
47          boolean closePosition(N accountId, TradeableInstrument<M>
            instrument);
48
49  }
```

If we create such specifications/interfaces for each aspect of the platform interaction, we can in theory create providers for these services and swap them when required, through the Spring configuration. From code organization perspective, all these interfaces, therefore, go in a project that forms part of the core API. This project will therefore be broker/exchange provider-agnostic and will comprise such interfaces and services.

- Write services that solve a single business problem or a collection of related problems. These services lend themselves to easy unit testability and code reuse that eventually leads to better software quality.
- Loosely couple services. This enables reducing system dependencies and as a result results in more maintainable software. Our software will be continuously evolving as one might decide to integrate more social media feeds or add more complex strategies. Writing loosely coupled components ensures that we have little knockon effect on already working code.

- High unit test coverage. It is extremely important that we aim to have a high unit test coverage. When used in a production environment where real money could be involved, large unit tests coverage will ensure that we catch regressions and bugs early on and prevent the manifestation of bugs as much as possible in the production environment.

Code Organization and Software Stack Used

Following on from our discussion of design goals in the previous sections, the code will be organized in at least three different projects. That means at least three JAR files will be produced from the build. Why are we saying at least three? Remember from our earlier discussion, that one of the key design goals is to be able to switch provider implementation at runtime. Since we could have more than one provider from which we can decide, there will be at least three JAR files (see Figure 1-1). We are going to discuss only one implementation in the book, i.e., the OANDA REST API implementation. Developers who use the framework are encouraged to develop more provider implementations:

- `trading-core` is the core of the project. It comprises all the specifications/interfaces that must be implemented. It also comprises all the generic services that use the core interfaces and provide additional useful API methods.
- `oanda-restapi` is our reference implementation for the specification and will be discussed in the book. You are more than welcome to swap this with your own.
- `tradingbot-app` is the main application that uses Spring to inject the provider API at runtime. It is also the project where we define our strategies and can implement app-specific stuff. Later in the book, we are going to talk about integration with social media, especially Twitter, which we will implement in this project.

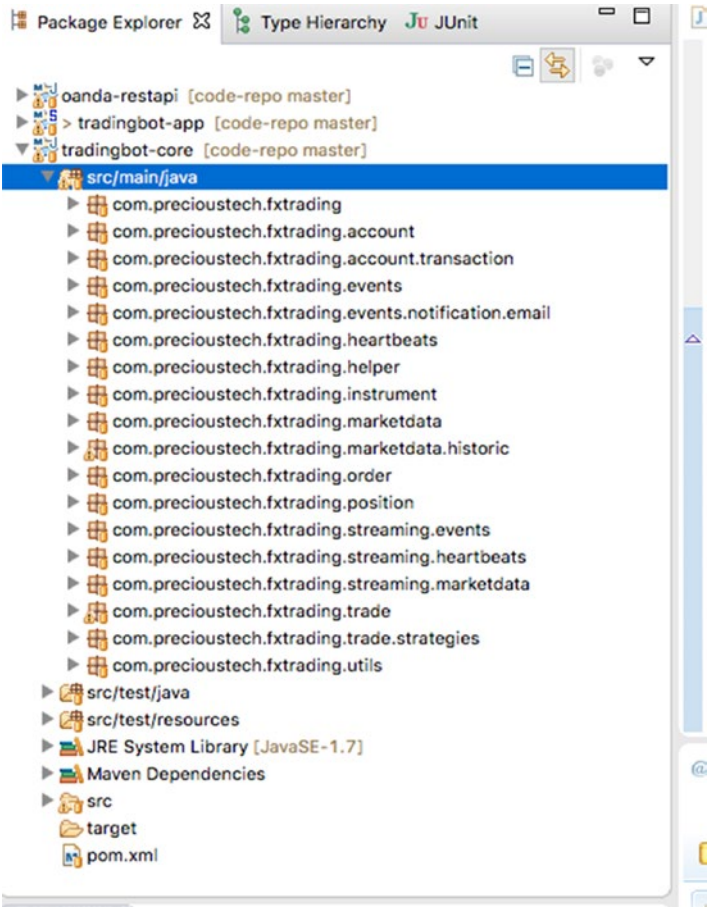

Figure 1-1. *Java projects*

To build our bot we are going to use the following set of software and tools:

- Java SDK 1.7+
- Spring Framework 4.1, Spring Social 1.1 (dependency in the `tradingbot-app` project only)
- Guava 18.0
- HttpClient 4.3
- Maven 3.2.5
- Eclipse IDE

OANDA REST API as Reference Implementation

■ **Note** I have no current or past commercial relationship with OANDA and I have chosen OANDA REST API as a reference implementation simply on its technical merit and the fact that it is likely to have wider adoption as it is free to use. If there is something similar around, I would be happy to give it a try as well.

I encountered the OANDA API by chance. At the time, I was looking for a broker who was offering a free API for trading (just in case I wanted to convert to a production account) and more importantly supported the Java programming language. It was very easy to get started, as most of the examples used curl to demonstrate various trading actions like getting a list of instruments, placing an order, or getting historical candlesticks data. I could just type the curl commands on my Mac terminal and fire away. I could easily see the JSON response received with each curl request fired. By seeing the responses and data in realtime, I got a really good idea and appreciation of various APIs supporting instruments, orders, rates, positions, etc.

The curl command examples was a good starting point and I started to experiment writing equivalent commands as test cases in Java described on the REST API[4] page.

Opening an OANDA Practice Account

Before you can start using the API, you must sign up for a free practice[5] account. When you head there and click on open an account button, you will see a signup screen like the one shown in Figure 1-2.

[4] http://developer.oanda.com/rest-live/introduction/
[5] http://fxtrade.oanda.com

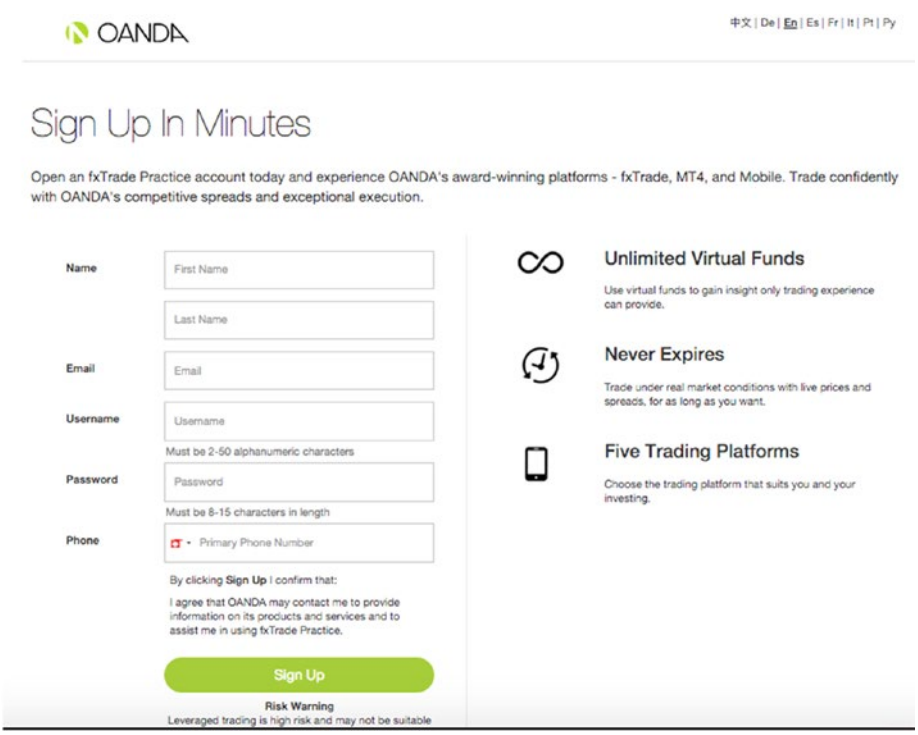

Figure 1-2. *Oanda account signup*

Once you have successfully signed up, you are now ready to sign in to your practice account. You will see the login screen shown in Figure 1-3 when you are ready to log in.