CHAPTER 12

Configuration, Deployment, and Running the Bot

We are now all set to put the bot into action. All that remains before pressing the button is the configuration and deployment. Configuration of the bot involves the following steps:

- Spring configuration of the core services and API, i.e., `tradingbot-app.xml`.
- Property file for the core Spring configuration, i.e., `tradingbot.properties`.
- Spring configuration of the runtime provider (OANDA in our case), i.e., `tradingbot-oanda.xml`.
- Property file for the runtime provider Spring configuration, i.e., `tradingbot-oanda.properties`.

Just like the JAR files, we also have a separate configuration file for the runtime provider. As we will see later in the chapter, the name of the provider configuration file is passed in as a program argument in order to easily pick the provider at runtime. Without further ado, let's jump into the configuration of the bot.

Configuring the Trading Bot

We start by creating the main configuration file in the `tradingbot-app` project in the `src/main/resources` folder, which is automatically included in the build and as a result ends up inside the `tradingbot-app.jar` file. The name of the file is `tradingbot-app.xml`. We also create the properties file, called `tradingbot.properties`, that has the values for various placeholders referenced in this Spring configuration file.

We also use the cool *SpEL*[1] in our configuration, the reason for which will become clear as we progress through this chapter.

[1] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html

S. Varshney, *Building Trading Bots Using Java*, DOI 10.1007/978-1-4842-2520-2_12

Core Beans Configuration

In this section, we configure all the core beans that are part of the core API except the services which we will configure later. These beans include e-mail, queues, trading config, etc. which are central to the functioning of the bot.

Since all our dependency beans are `@Autowired`, we switch on annotation processing by including `<context:annotation-config/>` in the configuration file. This will ensure that all the dependencies we have injected via `@Autowired` will be processed correctly. We also switch on property file processing by including the following in the configuration file and also specify where to find the properties file:

```
1   <context:property-placeholder location="classpath:tradingbot.properties"
    ignore-unresolvable="true"/>
```

The `trading.properties` has the following properties. We added some sample values to demonstrate how they look:

```
1   mail.to=tradingbot@gmail.com
2   mail.host=smtp.gmail.com
3   mail.port=465
4   mail.user=foobar@gmail.com
5   mail.password=foobar123
6   twitter.consumerKey=veJerryUYUDHUliDXjlC6o4KR
7   twitter.consumerSecret=L5iFptFbLKMyUoH8NB3yAodEeOEQMukriUt3mnNICEKy
    Np2OIh
8   twitter.accessToken=0123456789-LISP3YOOmLgDOSw44vgdVORr7313Zy
    PrAtOX3jU
9   twitter.accessTokenSecret=5IrbOYUlGOLd7UPeG6GPxFjVNWHREj4MN2
    TaoTL2kmVTT
```

We have already discussed the Twitter integration properties in Chapter 8. The `mail` properties are used to configure the `JavaMailSender` bean in the Spring configuration except `mail.to`, which actually is the `TO` address of the user who gets all the notifications generated by the bot.

We now turn our attention to the configuration of the core *beans* that make up the bot. The first is the configuration of the `TradingConfig` class. This class must be configured correctly for other services to work. We must also be careful to configure some of the values as they might affect the orders that are placed. For example, the `maxAllowedQuantity` is the maximum size of an order placed. In our case, it is just 5, but changing this value to, say, 50000 will place a huge order, which might lead to a margin call if the trade goes in the opposite direction.

Let's briefly discuss each of the config params in the `TradingConfig` class:

- `minReserveRatio` is the ratio of the amount available to trade to the total amount in the account. A value of 0.1 stipulates that if this amount falls below 10%, the bot stops putting in new orders.

- `maxAllowedNetContracts` is the maximum net the bot will go long or short on a given currency. This setting was inspired by the event on Jan 15, 2015, when SNB[2] unexpectedly ended the CHF peg versus EUR of 1.2 and CHF soured by nearly 30%. Anyone who was massively short CHF ended up losing huge amounts of money.
- `minAmountRequired` is the minimum balance in the designated currency of the account required to allow the bot to place orders. If the balance falls below this threshold, no further order placement is possible.
- `max10yrWmaOffset` defines the safe zone of a currency pair. We discussed it in detail while discussing `PreOrderValidationService`.
- `fadeTheMovePriceExpiry` is the time in minutes, the observation interval, for the `FadeTheMove` strategy.
- `fadeTheMoveJumpReqdToTrade` is the net jump in pips required during the `fadeTheMovePriceExpiry` that will create a scenario to place limit orders in the opposite direction of the market move.
- `fadeTheMoveDistanceToTrade` is the distance from the current price in pips when the limit order placed by the `FadeTheMoveStrategy` will be filled.
- `fadeTheMovePipsDesired` is the profit desired in pips for the `FadeTheMove` strategy.

```
<bean id="tradingConfig"
        class="com.precioustech.fxtrading.tradingbot.TradingConfig">
                <property name="minReserveRatio" value="0.1"/>
                <property name="maxAllowedQuantity" value="10"/>
                <property name="maxAllowedNetContracts" value="5"/>
                <property name="minAmountRequired" value="10.0"/>
                <property name="mailTo" value="${mail.to}"/>
                <property name="max10yrWmaOffset" value="0.1"/>
                <property name="fadeTheMoveJumpReqdToTrade" value="45"/>
                <property name="fadeTheMoveDistanceToTrade" value="25"/>
                <property name="fadeTheMovePipsDesired" value="10"/>
                <property name="fadeTheMovePriceExpiry" value="15"/>
</bean>
```

Next we configure the Google EventBus and all the callback handlers that use the EventBus to disseminate the event payloads. We also configure the bean post processor `FindEventBusSubscribers`, which we discussed in the first chapter. It finds all

[2] https://www.snb.ch/en/mmr/reference/pre_20150115/source/pre_20150115.en.pdf

classes with methods annotated by `@Subscriber` and registers them with the EventBus automatically.

```
<bean id="eventBus" class="com.google.common.eventbus.EventBus"/>
<bean id="findEventBusSubscribers"
  class="com.precioustech.fxtrading.tradingbot.spring.
  FindEventBusSubscribers"/>
<bean id="eventCallback"
  class="com.precioustech.fxtrading.events.EventCallbackImpl">
  <constructor-arg index="0" ref="eventBus"/>
</bean>
<bean id="heartBeatCallback"
   class="com.precioustech.fxtrading.heartbeats.HeartBeatCallbackImpl">
   <constructor-arg index="0" ref="eventBus"/>
</bean>
<bean id="marketEventCallback"
   class="com.precioustech.fxtrading.marketdata.MarketEventHandlerImpl">
   <constructor-arg index="0" ref="eventBus"/>
</bean>
```

For the `JavaMailSender` bean, the configuration looks like this:

```
<bean id="mailSender" class="org.springframework.mail.javamail.
JavaMailSenderImpl">
        <property name="host" value="${mail.host}"/>
        <property name="port" value="${mail.port}"/>
        <property name="username" value="${mail.user}"/>
        <property name="password" value="${mail.password}"/>
        <property name="javaMailProperties">
          <props>
              <prop key="mail.transport.protocol">smtps</prop>
              <prop key="mail.smtp.auth">true</prop>
              <prop key="mail.smtp.starttls.enable">true</prop>
              <prop key="mail.smtp.socketFactory.class">javax.net.ssl.
              SSLSocketFactory</prop>
              <prop key="mail.debug">false</prop>
              <prop key="mail.smtp.socketFactory.fallback">false</prop>
          </props>
        </property>
    </bean>
```

The placeholders for this bean are defined in the `tradingbot.properties` file. A related service that uses the `JavaMailSender` bean to send out notifications is configured as follows:

```
<bean id="eventEmailNotifier"
    class="com.precioustech.fxtrading.tradingbot.events.notification.
    email.EventEmailNotifier"/>
```

The orderQueue is used to place orders and has a very simple configuration.

```
1   <bean id="orderQueue" class="java.util.concurrent.LinkedBlockingQueue"/>
```

We use the built-in Spring Task Scheduler [^sprsched] to schedule our jobs. An example job is polling new tweets for our configured Twitter accounts.

```
1   <task:scheduler id="taskScheduler" pool-size="5"/>
```

Twitter-Related Beans Configuration

We first configure TwitterTemplate, which enables all interaction with Twitter and is part of Spring Social.

```
1   <bean id="twitter" class="org.springframework.social.twitter.api.impl.
    TwitterTemplate">
2                   <constructor-arg index="0" value="${twitter.
                    consumerKey}"/>
3                   <constructor-arg index="1" value="${twitter.
                    consumerSecret}"/>
4                   <constructor-arg index="2" value="${twitter.
                    accessToken}"/>
5                   <constructor-arg index="3" value="${twitter.
                    accessTokenSecret}"/>
6   </bean>
```

The placeholders for the configuration are all defined in the tradingbot.properties file, which we discussed in the previous section. It's pretty self-explanatory.

Next we configure a bean that is a list of all Twitter accounts for which we want to process the tweets. Although not a dependency for other beans, it is used by SpEL to inject values. For example:

```
 1   <bean id="fxTweeterList" class="java.util.ArrayList">
 2                   <constructor-arg index="0">
 3                           <list>
 4                                   <value>SignalFactory</value>
 5                                   <value>Forex_EA4U</value>
 6                           </list>
 7                   </constructor-arg>
 8   </bean>
 9   <util:map id="tweetHandlerMap">
10           <entry key="#{fxTweeterList[0]}">
11                   <bean  class=
12   "com.precioustech.fxtrading.tradingbot.social.twitter.tweethandler.
     SignalFactoryFXTweetHandler">
```

```
13                          <constructor-arg index="0" value="#{fxTweeter
                            List[0]}"/>
14                  </bean>
15          </entry>
16          <entry key="#{fxTweeterList[1]}">
17                  <bean class=
18          "com.precioustech.fxtrading.tradingbot.social.twitter.
            tweethandler.ZuluTrader101FXTweetHandler">
19                          <constructor-arg index="0" value="#{fxTweeter
                            List[1]}"/>
20                  </bean>
21          </entry>
22  </util:map>
```

The SpEL enables us to define all the Twitter accounts in once place and then reference the list via expressions and inject the value of these expressions into other beans.

If we need to listen to more Twitter accounts for tweets, we just create or reuse existing handlers (if applicable) and configure them here.

Provider Beans Configuration

In this section, we configure all the beans that implement the `Provider` interfaces and are provider-specific. Since our discussion throughout the book has been based on the OANDA REST API, we define it inside its own configuration called `tradingbot-oanda.xml` and which has all its properties defined in the `tradingbot-oanda.properties` file.

It is important that the ID given to the beans in this OANDA config file be reused in other provider implementations, since the main configuration references this ID to configure the core services bean.

We begin by looking at the properties file:

```
1  oanda.url=https://api-fxtrade.oanda.com
2  oanda.accessToken=7d741c1234f25d9f5a094e53a356789b-2c9a7b495789o4
   e177210af8e111c2f6
3  oanda.userName=foo
4  oanda.accountId=123456
5  oanda.streaming.url=https://stream-fxtrade.oanda.com
```

- `oanda.url` is the URL we want to point our trading bot to. Remember we have the live, practice, and sandbox environments provided by OANDA.
- `oanda.streaming.url` is the streaming URL for tick data and platform events.
- `oanda.accessToken` is the token that was generated in the environment pointed by `oanda.url`.

- `oanda.userName` is a valid username in the environment pointed by `oanda.url`.
- `oanda.accountId` is a valid account belonging to the value configured for `oanda.userName` in the environment pointed by `oanda.url`. A valid account ID is required to start a market data stream, for example.

We provide the location of this properties file, as usual as a *classpath* location, and turn on the annotation processing, like we did for the main configuration file.

```
1   <context:annotation-config/>
2   <context:property-placeholder location="classpath:tradingbot-oanda.
    properties"
3           ignore-unresolvable="true"/>
```

We now list the configurations for all the OANDA provider services.

AccountDataProviderService

```
1   <bean id="accountDataProvider" class=
2     "com.precioustech.fxtrading.oanda.restapi.account.
      OandaAccountDataProviderService">
3                   <constructor-arg index="0" value="${oanda.url}"/>
4                   <constructor-arg index="1" value="${oanda.userName}"/>
5                   <constructor-arg index="2" value="${oanda.accessToken}"/>
6   </bean>
```

ProviderHelper

```
1   <bean id="providerHelper"
2       class="com.precioustech.fxtrading.oanda.restapi.helper.
        OandaProviderHelper"/>
```

InstrumentDataProvider

```
1   <bean id="instrumentDataProvider"
2     class="com.precioustech.fxtrading.oanda.restapi.instrument.
      OandaInstrumentDataProviderService">
3                 <constructor-arg index="0" value="${oanda.url}"/>
4                 <constructor-arg index="1" value="${oanda.accountId}"/>
5                 <constructor-arg index="2" value="${oanda.
                  accessToken}"/>
6   </bean>
```

CurrentPriceInfoProvider

```
1     <bean id="currentPriceInfoProvider"
2       class="com.precioustech.fxtrading.oanda.restapi.marketdata.
        OandaCurrentPriceInfoProvider">
3                       <constructor-arg index="0" value="${oanda.url}"/>
4                       <constructor-arg index="1" value="${oanda.
                        accessToken}"/>
5   </bean>
```

HistoricMarketDataProvider

```
1   <bean id="historicMarketDataProvider"
2     class="com.precioustech.fxtrading.oanda.restapi.marketdata.historic.
      OandaHistoricMarketDataProvider">
3                       <constructor-arg index="0" value="${oanda.url}"/>
4                       <constructor-arg index="1" value="${oanda.
                        accessToken}"/>
5   </bean>
```

OrderManagementProvider

```
1   <bean id="orderManagementProvider"
2     class="com.precioustech.fxtrading.oanda.restapi.order.
      OandaOrderManagementProvider">
3                       <constructor-arg index="0" value="${oanda.url}"/>
4                       <constructor-arg index="1" value="${oanda.
                        accessToken}"/>
5                       <constructor-arg index="2" ref="accountDataProvider"/>
6   </bean>
```

TradeManagementProvider

```
1   <bean id="tradeManagementProvider"
2     class="com.precioustech.fxtrading.oanda.restapi.trade.
      OandaTradeManagementProvider">
3                       <constructor-arg index="0" value="${oanda.url}"/>
4                       <constructor-arg index="1" value="${oanda.
                        accessToken}"/>
5             </bean>
```

PositionManagementProvider

```
<bean id="positionManagementProvider"
  class="com.precioustech.fxtrading.oanda.restapi.position.
  OandaPositionManagementProvider">
                <constructor-arg index="0" value="${oanda.url}"/>
                <constructor-arg index="1" value="${oanda.
                accessToken}"/>
</bean>
```

EventStreamingService

```
<bean id="eventsStreamingService"
  class="com.precioustech.fxtrading.oanda.restapi.streaming.events.
  OandaEventsStreamingService">
              <constructor-arg index="0" value="${oanda.streaming.
              url}"/>
              <constructor-arg index="1" value="${oanda.
              accessToken}"/>
              <constructor-arg index="2" ref="accountDataProvider"/>
              <constructor-arg index="3" ref="eventCallback"/>
              <constructor-arg index="4" ref="heartBeatCallback"/>
              <constructor-arg index="5" value="EVENTSTREAM"/>
</bean>
```

We assign the source ID `EVENTSTREAM` as a heartbeat source ID, since the same class is responsible for handling the event heartbeats.

Platform Event Handlers

```
<bean id="orderEventHandler"
  class="com.precioustech.fxtrading.oanda.restapi.events.
  OrderFilledEventHandler">
        <constructor-arg index="0" ref="tradeInfoService"/>
</bean>
<bean id="tradeEventHandler"
  class="com.precioustech.fxtrading.oanda.restapi.events.
  TradeEventHandler">
        <constructor-arg index="0" ref="tradeInfoService"/>
</bean>
```

These event handlers need to be assigned to platform events that they will handle:

```
<util:map id="eventEmailContentGeneratorMap" key-type="com.
precioustech.fxtrading.events.Event">
        <entry
  key="#{T(com.precioustech.fxtrading.oanda.restapi.events.
  OrderEvents).MARKET_ORDER_CREATE}"
  value-ref="orderEventHandler"/>
        <entry
  key="#{T(com.precioustech.fxtrading.oanda.restapi.events.
  OrderEvents).LIMIT_ORDER_CREATE}"
  value-ref="orderEventHandler"/>
        <entry
  key="#{T(com.precioustech.fxtrading.oanda.restapi.events.
  OrderEvents).ORDER_CANCEL}"
  value-ref="orderEventHandler"/>
        <entry
  key="#{T(com.precioustech.fxtrading.oanda.restapi.events.
  OrderEvents).ORDER_FILLED}"
  value-ref="orderEventHandler"/>
        <entry
  key="#{T(com.precioustech.fxtrading.oanda.restapi.events.
  TradeEvents).TRADE_CLOSE}"
  value-ref="tradeEventHandler"/>
        <entry
  key="#{T(com.precioustech.fxtrading.oanda.restapi.events.
  TradeEvents).STOP_LOSS_FILLED}"
  value-ref="tradeEventHandler"/>
        <entry
  key="#{T(com.precioustech.fxtrading.oanda.restapi.events.
  TradeEvents).TAKE_PROFIT_FILLED}"
  value-ref="tradeEventHandler"/>
</util:map>
```

We configure the `eventEmailContentGeneratorMap` map to express interest in notifications that we want the bot to generate when the platform event is generated. The corresponding handlers are responsible for generating the content for the e-mail. as discussed in the previous chapter.

MarketDataStreamingService

```
<bean id="marketDataStreamingService"
  class="com.precioustech.fxtrading.oanda.restapi.streaming.marketdata.
  OandaMarketDataStreamingService">
                <constructor-arg index="0" value="${oanda.streaming.
                url}"/>
```

```
4                   <constructor-arg index="1" value="${oanda.
                    accessToken}"/>
5                   <constructor-arg index="2" value="${oanda.accountId}"/>
6                   <constructor-arg index="3" ref="tradeableInstrument
                    List"/>
7                   <constructor-arg index="4" ref="marketEventCallback"/>
8                   <constructor-arg index="5" ref="heartBeatCallback"/>
9                   <constructor-arg index="6" value="MKTDATASTREAM"/>
10  </bean>
```

We assign the source ID `MKTDATASTREAM` as a heartbeat source ID, since the same class is responsible to tick data heartbeats.

The `tradeableInstrumentList` is the list of instruments for which we want to subscribe tick data from the OANDA platform. The list is configured as follows:

```
1   <bean id="tradeableInstrumentList" class="java.util.ArrayList">
2           <constructor-arg index="0">
3                   <list>
4                           <bean class="com.precioustech.fxtrading.
                            instrument.TradeableInstrument">
5           <constructor-arg  index="0" value="USD_CAD"/>
6         </bean>
7                           <bean class="com.precioustech.fxtrading.
                            instrument.TradeableInstrument">
8           <constructor-arg index="0" value="GBP_USD"/>
9         </bean>
10                          <bean class="com.precioustech.fxtrading.
                            instrument.TradeableInstrument">
11          <constructor-arg index="0" value="AUD_JPY"/>
12        </bean>
13                          <bean class="com.precioustech.fxtrading.
                            instrument.TradeableInstrument">
14          <constructor-arg index="0" value="EUR_NZD"/>
15        </bean>
16                          <bean class="com.precioustech.fxtrading.
                            instrument.TradeableInstrument">
17          <constructor-arg index="0" value="GBP_CHF"/>
18        </bean>
19                          <bean class="com.precioustech.fxtrading.
                            instrument.TradeableInstrument">
20          <constructor-arg  index="0" value="EUR_JPY"/>
21        </bean>
22                  </list>
23          </constructor-arg>
24  </bean>
```

This concludes our discussion of the configuration of all the *provider* beans for the OANDA implementation.

Strategies Configuration

In this book we discussed a couple of strategies that happen to get activated via a scheduler-based invocation. In this section we configure them and show how they are invoked.

```
<bean id="fadeTheMoveStrategy"
    class="com.precioustech.fxtrading.tradingbot.strategies.
    FadeTheMoveStrategy">
        <constructor-arg index="0" ref="tradeableInstrumentList"/>
</bean>
<bean id="copyTwitterStrategy"
    class="com.precioustech.fxtrading.tradingbot.strategies.
    CopyTwitterStrategy"/>
```

The configuration is fairly straightforward. The `FadeTheMove` strategy has an additional constructor dependency that gets injected with a list of instruments to observe and place an order if the given conditions are met.

Now we look at the scheduler configuration that invokes a given strategy bean method after every time interval `T`:

```
<task:scheduled-tasks scheduler="taskScheduler">
                  <task:scheduled ref="fadeTheMoveStrategy"
                  method="analysePrices"
                fixed-delay="60000"/>
                  <task:scheduled ref="copyTwitterStrategy"
                  method="harvestAndTrade"
                fixed-delay="300000"/>
</task:scheduled-tasks>
```

The scheduler invokes every 60000ms the `analysePrices()` method of the `fadeTheMoveStrategy` bean. It also invokes every 300000ms the `harvestAndTrade()` method of the `copyTwitterStrategy` bean.

Services Configuration

In this concluding section on configuration, we discuss how to configure the core services of the bot. These services could be considered the public API of the bot if there were a GUI client or for that matter any other client interacting with the bot. Most of the demo programs in this book directly interact with these services.

AccountInfoService

```
<bean id="accountInfoService"
    class="com.precioustech.fxtrading.account.AccountInfoService">
        <constructor-arg index="0" ref="accountDataProvider"/>
```

```
        <constructor-arg index="1" ref="currentPriceInfoProvider"/>
        <constructor-arg index="2" ref="tradingConfig"/>
        <constructor-arg index="3" ref="providerHelper"/>
</bean>
```

InstrumentService

```
<bean id="instrumentService"
    class="com.precioustech.fxtrading.instrument.InstrumentService">
                <constructor-arg index="0" ref="instrumentDataProvider"/>
</bean>
```

MovingAverageCalculationService

```
<bean id="movingAverageCalculationService"
  class="com.precioustech.fxtrading.marketdata.historic.
  MovingAverageCalculationService">
                <constructor-arg index="0" ref="historicMarketData
                Provider"/>
</bean>
```

OrderInfoService

```
<bean id="orderInfoService"
    class="com.precioustech.fxtrading.order.OrderInfoService">
        <constructor-arg index="0" ref="orderManagementProvider"/>
</bean>
```

TradeInfoService

```
<bean id="tradeInfoService"
    class="com.precioustech.fxtrading.trade.TradeInfoService">
        <constructor-arg index="0" ref="tradeManagementProvider"/>
        <constructor-arg index="1" ref="accountInfoService"/>
</bean>
```

PreOrderValidationService

```
<bean id="preOrderValidationService"
     class="com.precioustech.fxtrading.order.PreOrderValidationService">
       <constructor-arg index="0" ref="tradeInfoService"/>
       <constructor-arg  index="1" ref="movingAverageCalculationService"/>
```

```
        <constructor-arg index="2" ref="tradingConfig"/>
        <constructor-arg index="3" ref="orderInfoService"/>
</bean>
```

OrderExecutionService

```
<bean id="orderExecutionService"
    class="com.precioustech.fxtrading.order.OrderExecutionService">
        <constructor-arg index="0" ref="orderQueue"/>
        <constructor-arg index="1" ref="accountInfoService"/>
        <constructor-arg index="2" ref="orderManagementProvider"/>
        <constructor-arg index="3" ref="tradingConfig"/>
        <constructor-arg index="4" ref="preOrderValidationService"/>
        <constructor-arg index="5" ref="currentPriceInfoProvider"/>
</bean>
```

DefaultHeartBeatService

```
<bean id="heartBeatService"
    class="com.precioustech.fxtrading.heartbeats.
    DefaultHeartBeatService">
        <constructor-arg index="0">
                <list>
                        <ref bean="eventsStreamingService"/>
                        <ref bean="marketDataStreamingService"/>
                </list>
        </constructor-arg>
</bean>
```

Building the Bot

The bot can easily be built using maven. Since there are three projects that need to be built in a given order, we can script the actions, instead of having to remember the build order. If you have cloned the code repository from GitHub, there is a `buildbot.bsh` file in the `<repo-code>/java` directory that will build the bot using maven. Figure 12-1 shows my bash terminal after having cloned the repository.

```
[ShekharMacBook:java shekhar$ pwd
/Volumes/05/git/book-code/java
[ShekharMacBook:java shekhar$ ls -l
total 8
-rwxr-xr-x   1 shekhar  staff  324 31 Jan 16:43 buildbot.bsh
drwxr-xr-x   9 shekhar  staff  306 31 Jan 17:32 oanda-restapi
drwxr-xr-x  11 shekhar  staff  374 31 Jan 17:32 tradingbot-app
drwxr-xr-x   9 shekhar  staff  306 31 Jan 17:32 tradingbot-core
drwxr-xr-x  10 shekhar  staff  340 27 Jan 20:10 tradingbot-demo-programs
ShekharMacBook:java shekhar$
```

Figure 12-1. *Code repo structure*

Before running the code, it is assumed that maven and Java 1.7 are installed and configured. When you type the `mvn --version` is typed command on the command line, you'll see output similar to what's shown in Figure 12-2.

```
ShekharMacBook:java shekhar$ mvn --version
Apache Maven 3.2.5 (12a6b3acb947671f09b81f49094c53f426d8cea1; 2014-12-14T18:29:23+01:00)
Maven home: /usr/local/Cellar/maven/3.2.5/libexec
Java version: 1.7.0_71, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.7.0_71.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.11", arch: "x86_64", family: "mac"
ShekharMacBook:java shekhar$
```

Figure 12-2. *mvn –version output*

If this is not the case, make sure maven is properly installed[3].

We are now ready to run the `buildbot.bsh` script. Before that, let's take a quick look at the script code:

```
1   function buildmodule {
2       cd $SCRIPT_DIR/$1
3       mvn clean install
4       if [[ "$?" -ne 0 ]]; then
5         echo "ERROR: $1 project build failed. BUILD FAILED"; exit -1;
6       fi
7   }
8   SCRIPT_DIR=`pwd`
9   buildmodule tradingbot-core
10  buildmodule oanda-restapi
11  buildmodule tradingbot-app
12
13  echo "TRADING BOT built successfully"
```

[3] https://maven.apache.org/install.html

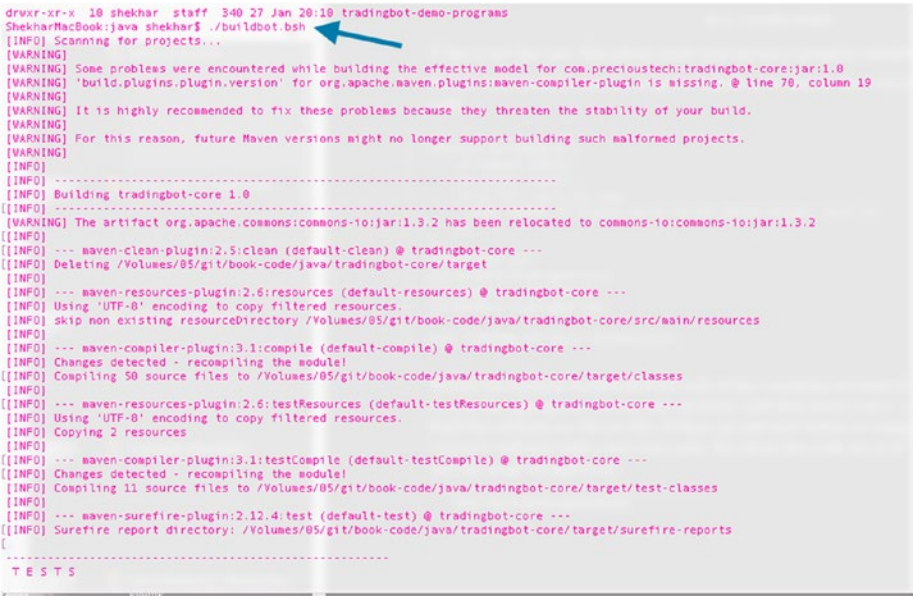

Figure 12-3. *Start build at the command line*

We call the function `buildmodule` for each of the three modules we want to build. The function first does a `cd` to the appropriate source code directory and then issues a `mvn clean install` command to build the artifacts (i.e., the *jar* file). If there are unit tests failure or compilation issues, `mvn clean install` will exit with a non-zero code. We check this code set in the `$?` variable, and if it's non-zero, we exit the script straightaway. See Figure 12-4.

Figure 12-4. *Build failure*

If the build succeeds, we should see the output in Figure 12-5.

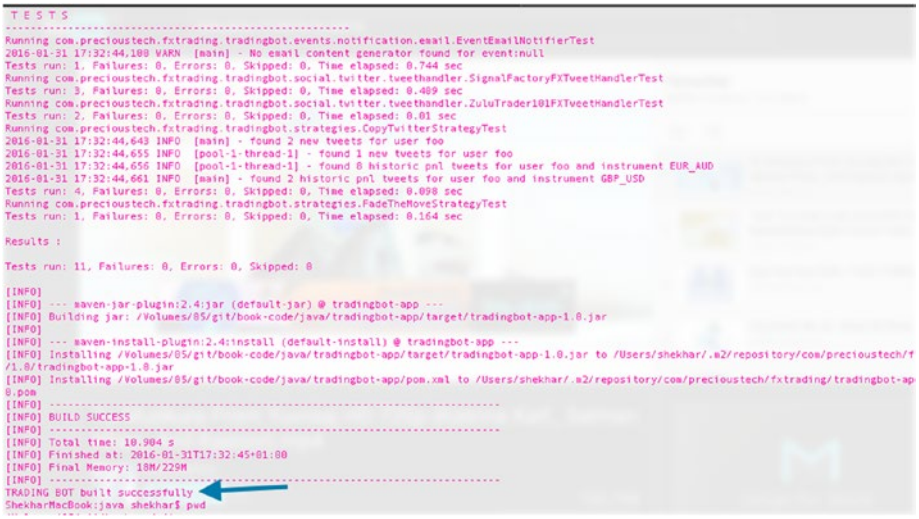

Figure 12-5. *Build success*

Running the Bot

■ **Warning** Serious losses can be incurred if some of the configuration values used to run the bot are extremely high. The author has taken every opportunity to highlight those throughout the book. In no event is the author liable for any consequential damages arising from the configuration values the user deems fit for his/her risk appetite.

Now that the bot is fully built, we are ready to run it. But before we really push the button, we need to make sure the following config params have been adapted and a build is done so that these changes are in the JAR file for the given user. Here is the full checklist of changes:

- All OANDA-specific parameters in `tradingbot-oanda.properties` belong to the user of a valid live or practice account.
- All Twitter-specific access tokens in `tradingbot.properties` are correctly populated. These valid tokens are generated for a Twitter app or for a Twitter account to which the user has full access.

- Double-check that the values for the `TradingConfig` bean are within comfortable limits, especially `maxAllowedQuantity`, `minReserveRatio`, and `maxAllowedNetContracts`.
- Configure a valid mail account in `tradingbot.properties`. The credentials are used to send e-mail notifications to the `mail.to` e-mail address.
- Configure `log4j.properties` in the `tradingbot-app` resources directory if required.
- Execute `buildbot.sh` again to rebuild the bot with the new set of parameters.

To run the bot with the strategies discussed in the book and OANDA implementation, we simply do the following

```
1   $ ./runbot-oanda.bsh
```

If all goes well, you should see the output shown in Figure 12-6 coming from the bot.

Figure 12-6. *Running the bot*

While the bot is running, we should see it harvesting the tweet shown in Figure 12-7 from time to time.

Figure 12-7. *New tweet harvested*

That is it. Our bot is now fully up and running. We should see that `FadeTheMove` strategy is very busy, especially around market moving currency events such as FOMC decision or GDP data from various countries. One point to bear in mind is that this strategy maintains an internal cache for the last 15 minutes. The more instruments you configure for subscription to tick the data stream, the more memory footprint you will have. Therefore, it may be imperative to run with a higher memory value. Since the bot runs inside maven, you would have to change the `MAVEN_OPTS` params.