# Table of Contents

Preface ........................................................................................................... viii
1. Document Conventions ........................................................................ viii
   1.1. Typographic Conventions ............................................................ viii
   1.2. Pull-quote Conventions ............................................................... ix
   1.3. Notes and Warnings .................................................................. x
1. Introduction ............................................................................................ 1
2. Building AlgoTrader ............................................................................... 2
   2.1. Command Line ........................................................................ 2
       2.1.1. Git Checkout ..................................................................... 2
       2.1.2. Maven Build .................................................................. 2
       2.1.3. Docker Build .................................................................. 2
   2.2. Eclipse ...................................................................................... 3
       2.2.1. Git Checkout ..................................................................... 3
       2.2.2. Maven Build .................................................................. 4
       2.2.3. Docker Build .................................................................. 4
   2.3. AlgoTrader UI ........................................................................... 4
       2.3.1. Git Checkout ..................................................................... 4
       2.3.2. Maven Build .................................................................. 5
       2.3.3. Docker Build .................................................................. 5
3. Domain Model ....................................................................................... 6
   3.1. Security Visitors ........................................................................ 6
   3.2. Data access objects (DAOs) ....................................................... 6
   3.3. Services ..................................................................................... 9
       3.3.1. Private Services ............................................................... 9
       3.3.2. Order Services ............................................................... 9
       3.3.3. Market Data Services .................................................... 10
   3.3.4. Historical Data Services .................................................... 10
   3.3.5. Reference Data Services .................................................... 10
4. Java Environment ................................................................................ 11
   4.1. AlgoTrader Project Structure .................................................. 11
       4.1.1. common project ............................................................ 11
       4.1.2. core project .................................................................. 12
       4.1.3. conf project .................................................................. 12
       4.1.4. launch project ................................................................ 12
       4.1.5. strategy projects .......................................................... 13
   4.2. Java Packages & Classes ......................................................... 13
4.3. Maven Environment ........................................................................ 13
   4.3.1. Maven assemblies ............................................................ 14
   4.3.2. Packaging strategies ......................................................... 14
5. Code Generation ................................................................................ 16
6. Database ............................................................................................. 18
   6.1. Database scripts ........................................................................ 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Transaction Handling</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Market Data</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>Adapters</td>
<td>21</td>
</tr>
<tr>
<td>8.1</td>
<td>Bloomberg</td>
<td>21</td>
</tr>
<tr>
<td>8.2</td>
<td>Currenex</td>
<td>21</td>
</tr>
<tr>
<td>8.3</td>
<td>DukasCopy</td>
<td>21</td>
</tr>
<tr>
<td>8.4</td>
<td>Exante</td>
<td>22</td>
</tr>
<tr>
<td>8.5</td>
<td>EzeSoft/RealTick</td>
<td>22</td>
</tr>
<tr>
<td>8.6</td>
<td>Fix Interface</td>
<td>22</td>
</tr>
<tr>
<td>8.7</td>
<td>Fortex</td>
<td>23</td>
</tr>
<tr>
<td>8.8</td>
<td>FXCM</td>
<td>24</td>
</tr>
<tr>
<td>8.9</td>
<td>IB Native Interface</td>
<td>24</td>
</tr>
<tr>
<td>8.10</td>
<td>JP Morgan</td>
<td>25</td>
</tr>
<tr>
<td>8.11</td>
<td>LMAX</td>
<td>25</td>
</tr>
<tr>
<td>8.12</td>
<td>Nexus Prime</td>
<td>25</td>
</tr>
<tr>
<td>8.13</td>
<td>PrimeXM</td>
<td>25</td>
</tr>
<tr>
<td>8.14</td>
<td>Quandl</td>
<td>26</td>
</tr>
<tr>
<td>8.15</td>
<td>QuantHouse</td>
<td>26</td>
</tr>
<tr>
<td>8.16</td>
<td>SocGen</td>
<td>26</td>
</tr>
<tr>
<td>8.17</td>
<td>Trading Technologies</td>
<td>26</td>
</tr>
<tr>
<td>8.18</td>
<td>UBS</td>
<td>27</td>
</tr>
<tr>
<td>8.19</td>
<td>Binance</td>
<td>27</td>
</tr>
<tr>
<td>8.20</td>
<td>Bitfinex</td>
<td>28</td>
</tr>
<tr>
<td>8.21</td>
<td>Bitflyer</td>
<td>28</td>
</tr>
<tr>
<td>8.22</td>
<td>BitMEX</td>
<td>28</td>
</tr>
<tr>
<td>8.23</td>
<td>Bitstamp</td>
<td>29</td>
</tr>
<tr>
<td>8.24</td>
<td>CoinAPI</td>
<td>29</td>
</tr>
<tr>
<td>8.25</td>
<td>Coinbase</td>
<td>30</td>
</tr>
<tr>
<td>8.26</td>
<td>Coinigy</td>
<td>30</td>
</tr>
<tr>
<td>8.27</td>
<td>CoinMarketCap</td>
<td>31</td>
</tr>
<tr>
<td>9</td>
<td>Execution Algos</td>
<td>32</td>
</tr>
<tr>
<td>9.1</td>
<td>Development of Execution Algos</td>
<td>32</td>
</tr>
<tr>
<td>9.2</td>
<td>Execution Algos entry form generation</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>Spring Services</td>
<td>35</td>
</tr>
<tr>
<td>10.1</td>
<td>Wiring Factories</td>
<td>35</td>
</tr>
<tr>
<td>10.2</td>
<td>Application Context</td>
<td>35</td>
</tr>
<tr>
<td>10.3</td>
<td>Abstract Services</td>
<td>37</td>
</tr>
<tr>
<td>10.4</td>
<td>Service initialization order</td>
<td>38</td>
</tr>
<tr>
<td>11</td>
<td>Events and Messaging</td>
<td>39</td>
</tr>
<tr>
<td>11.1</td>
<td>Embedded ActiveMQ message broker</td>
<td>39</td>
</tr>
<tr>
<td>11.2</td>
<td>Embedded Jetty HTTP server</td>
<td>39</td>
</tr>
<tr>
<td>11.3</td>
<td>RESTful interface</td>
<td>39</td>
</tr>
<tr>
<td>11.4</td>
<td>Event Dispatcher</td>
<td>40</td>
</tr>
<tr>
<td>11.5</td>
<td>Event Listeners</td>
<td>41</td>
</tr>
</tbody>
</table>
11.6. JMS Destinations ................................................................. 43
12. Configuration and Preferences API .................................................. 44
  12.1. Config Providers ................................................................. 44
  12.2. Config Beans ................................................................. 44
  12.3. Config Locator ............................................................... 46
13. Processes and Networking .......................................................... 47
  13.1. Processes ................................................................. 47
  13.2. Sockets ................................................................. 47
  13.3. RMI ................................................................. 48
14. Hibernate Sessions and Caching .................................................... 49
  14.1. Hibernate Caching ......................................................... 49
    14.1.1. Level-Zero Cache .................................................... 49
15. Logging ................................................................. 52
  15.1. Custom UI Log Event Appender ............................................. 52
List of Figures

2.1. Eclipse default JRE .............................................................................................................. 4
4.1. Maven Dependencies ........................................................................................................... 13
List of Tables

3.1. Private Services ................................................................. 9
4.1. common project .................................................................. 12
4.2. core project ...................................................................... 12
4.3. conf project ..................................................................... 12
4.4. launch project .................................................................. 12
4.5. strategy projects .............................................................. 13
5.1. Hibernate mapping files - custom meta attributes ............... 17
8.1. Bloomberg Infrastructure .................................................. 21
8.2. Currenex Infrastructure ...................................................... 21
8.3. DukasCopy Infrastructure .................................................. 21
8.4. Exante Infrastructure ......................................................... 22
8.5. EzeSoft/RealTick Infrastructure ........................................... 22
8.6. Fix Infrastructure ............................................................... 22
8.7. Fortex Infrastructure .......................................................... 23
8.8. FXCM Infrastructure .......................................................... 24
8.9. IB Infrastructure ............................................................... 24
8.10. JP Morgan Infrastructure .................................................... 25
8.11. LMAX Infrastructure ........................................................ 25
8.12. Nexus Prime Infrastructure ............................................... 25
8.13. PrimeXM Infrastructure .................................................... 26
8.14. Quandl Infrastructure ....................................................... 26
8.15. QuantHouse Infrastructure ............................................... 26
8.16. SocGen Infrastructure ........................................................ 26
8.17. Trading Technologies Infrastructure ..................................... 27
8.18. UBS Infrastructure ............................................................ 27
8.19. Main service classes ........................................................ 27
8.20. Main classes .................................................................. 27
8.21. Main service classes ........................................................ 28
8.22. Main classes .................................................................. 28
8.23. Main service classes ........................................................ 28
8.24. Main classes .................................................................. 28
8.25. Main service classes ........................................................ 28
8.26. Main classes .................................................................. 29
8.27. Main service classes ........................................................ 29
8.28. Main classes .................................................................. 29
8.29. Main service classes ........................................................ 29
8.30. Main classes .................................................................. 30
8.31. Main service classes ........................................................ 30
8.32. Main service classes ........................................................ 30
8.33. Main classes .................................................................. 30
8.34. Main service classes ........................................................ 31
8.35. Main classes .................................................................. 31
10.1. Bean Reference Factories ................................................................. 35
10.2. Application Context Files .............................................................. 36
10.3. Application Context References ........................................................ 36
11.1. Event Recipients .............................................................. 40
11.2. Standard event listener classes ......................................................... 41
13.1. Services and Processes ................................................................. 47
13.2. Sockets ......................................................................................... 47
Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the Liberation Fonts set. The Liberation Fonts set is also used in HTML editions. If not, alternative but equivalent typefaces are displayed.

1.1. Typographic Conventions

The following typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

System input, including shell commands, file names and paths, and key caps and key-combinations are presented as follows.

To see the contents of the file my_next_bestselling_novel in the current working directory, enter the cat my_next_bestselling_novel command at the shell prompt and press Enter to execute the command.

The above includes a file name, a shell command and a key cap, all distinguishable thanks to context.

Key-combinations can be distinguished from key caps by the symbol connecting each part of a key-combination. For example:

Press Enter to execute the command.

Press Ctrl-Alt-F1 to switch to the first virtual terminal. Press Ctrl-Alt-F7 to return to the X-Windows session.

The first sentence highlights the particular key cap to press. The second highlights two sets of three key caps, each set pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph are presented as follows.

File-related classes include filesystem for file systems, file for files, and dir for directories. Each class has its own associated set of permissions.

Words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles are presented as follows.

Choose System → Preferences → Mouse from the main menu bar to launch Mouse Preferences. In the Buttons tab, click the Left-handed mouse check box and click Close to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

1 https://pagure.io/liberation-fonts
To insert a special character into a gedit file, choose Applications → Accessories → Character Map from the main menu bar. Next, choose Search → Find from the Character Map menu bar, type the name of the character in the Search field and click Next. The character sought will be highlighted in the Character Table. Double-click this highlighted character to place it in the Text to copy field and then click the Copy button. Now switch back to the document and choose Edit → Paste from the gedit menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all distinguishable by context.

Note the shorthand used to indicate traversal through a menu and its sub-menus. This is to avoid the difficult-to-follow ‘Select Mouse from the Preferences sub-menu in the System menu of the main menu bar’ approach.

Italics denotes text that does not need to be imputed literally or displayed text that changes depending on circumstance. Replaceable or variable text is presented as follows.

To connect to a remote machine using ssh, type ssh username@domain.name at a shell prompt. If the remote machine is example.com and the username on that machine is john, type ssh john@example.com.

The mount -o remount file-system command remounts the named file system. For example, to remount the home file system, the command is mount -o remount /home.

To see the version of a currently installed package, use the rpm -q package command. It will return a result as follows: package-version-release.

Note the words in italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text entered when issuing a command or for text displayed by the system.

1.2. Pull-quote Conventions

Two commonly multi-line data types are set off visually from the surrounding text.

Output sent to a terminal is presented as follows:

```
books        Desktop   documentation  drafts  mss    photos   stuff  git
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are presented and highlighted as follows:

```
package org.jboss.book.jca.ex1;
import javax.naming.InitialContext;
public class ExClient {
    public static void main(String args[]) throws Exception {
```
InitialContext iniCtx = new InitialContext();
Object ref = iniCtx.lookup("EchoBean");
EchoHome home = (EchoHome) ref;
Echo echo = home.create();

System.out.println("Created Echo");

System.out.println("Echo.echo('Hello') = " + echo.echo("Hello");
}
Introduction

This document provides additional information on the internal implementation of AlgoTrader for cases when clients wish to make changes to the platform or extends its functionality.

Note

A source code license is required to update the internal parts of AlgoTrader.
Building AlgoTrader

AlgoTrader can be built from its source either via command line or via Eclipse

Note
AlgoTrader based trading strategies can be developed and started without building AlgoTrader first

2.1. Command Line

To build AlgoTrader via command line please perform the following steps.

2.1.1. Git Checkout

If one hasn’t installed git, please refer to git installation in the Reference Documentation (chapter 2.1.1. Prerequisites)

Perform a Git clone from the command line:

```shell
git clone https://gitlab.algotrader.ch/main/algotrader.git
```

Note
User name and password will be provided when signing up for an AlgoTrader license

2.1.2. Maven Build

Execute the following maven command to build all maven projects

```shell
mvn clean install
```

Note
When running the build process for the first time, this will take a few minutes since all maven dependencies have to be downloaded.

2.1.3. Docker Build

Execute the following Docker command to build the AlgoTrader Docker image:
2.2. Eclipse

To build AlgoTrader from within Eclipse please follow this process.

2.2.1. Git Checkout

- Inside Eclipse switch to the Java Perspective (Windows --> Open Perspective --> Java)
- Click File / Import / Git / Projects from Git / Clone URI
- Select the following URI https://gitlab.algotrader.ch/main/algotrader.git
- Enter User and Password (provided when licensing AlgoTrader)
- Click Next
- Select master
- Click Next
- Select Import existing projects and click Next
- Select the project algotrader
- Select the new project algotrader, right click and select Import / Maven / Existing Maven Projects and select:
  - common
  - core
  - conf
  - launch
- Click Finish

This will result in the following Eclipse projects:

- algotrader-common
- algotrader-core
- algotrader-conf
- algotrader-launch

1 https://gitlab.algotrader.ch/main/algotrader.git
Note

The compilation will show errors, which should go away after the next section has been completed.

2.2.2. Maven Build

Before running the maven build from within Eclipse please make sure that the default Eclipse Java runtime environment is pointing to a Java JDK. To verify please go to Window / Preferences / Java / Installed JREs. If the default JRE is pointing to a Java JRE, then please add a reference to the Java JDK.

Figure 2.1. Eclipse default JRE

To generate the code right click on the project `algotrader` inside Eclipse and select / Run As / Maven install. This will generate all maven modules.

Now refresh all projects. Eclipse will compile all java code automatically. In case there is an error message Project configuration is not up-to-date with pom.xml... on any of the projects please select: Maven->Update Project from the project context menu.

2.2.3. Docker Build

The AlgoTrader Docker Image needs to be built from the command line (see above).

2.3. AlgoTrader UI

2.3.1. Git Checkout

Perform a Git clone from the command line:
git clone git@gitlab.algotrader.ch:main/HTML5-UI.git

2.3.2. Maven Build

Execute the following maven command to build UI through maven

```shell
mvn clean install -DALGOTRADER_NPM_NEXUS_AUTH=auth_token
```

*auth_token* is a *username:password* for AlgoTrader Nexus repository transformed by base64 function. That transformation can for example be done by using Window Powershell:

```powershell
[Convert]::ToBase64String($Bytes)
```

2.3.3. Docker Build

After the Maven build of the UI, rebuild the back end Docker image (see above).
Domain Model

3.1. Security Visitors

The Visitor Pattern\(^1\) is a way of separating an algorithm from an object structure on which it operates. Using this pattern it is possible to implement custom Logic per Entity without polluting the Entity code itself.

AlgoTrader provides the interface `ch.algotrader.visitor.EntityVisitor` which must be implemented by all Entity Visitors. Each Entity Visitor has two generic type parameters R and P. R is the return type (or `java.lang.Void`) returned by all visit methods and P is an arbitrary parameter object that can be added to the visit methods.

In addition there is the `ch.algotrader.visitor.PolymorphicEntityVisitor` which reflects the entire inheritance tree of all Securities. For example if there is no `visitFuture` method the `PolymorphicEntityVisitor` will automatically invoke the `visitSecurity` method.

The accept method of each Entity can be used to process an arbitrary Visitor like this:

```
entity.accept(MyVisitor.INSTANCE);
```

In AlgoTrader there are two Visitors available which are used by the AlgoTrader Server

- **InitializingVisitor**
  - Is used to make sure certain Hibernate Entity References are initialized / loaded.

- **ScalingVisitor**
  - Is used to scale quantities and prices

- **SecurityVolumeVisitor**
  - Is used to determine if a particular Security is supposed to report volumes

- **TickValidationVisitor**
  - Used to validate a Tick by rules defined per Security

3.2. Data access objects (DAOs)

The AlgoTrader DAO framework of consists of several main components

- **BaseEntityI**
  - `BaseEntityI` represents an abstract serializable persistent entity with a synthetic identifier of type long.

\(^1\) [https://en.wikipedia.org/wiki/Visitor_pattern](https://en.wikipedia.org/wiki/Visitor_pattern)
Data access objects (DAOs)

ReadOnlyDao
ReadOnlyDao represents an interface for common retrieval operations for entity classes.

ReadWriteDao
ReadWriteDao extends ReadOnlyDao and represents an interface for common retrieval and mutation operations.

AbstractDao
AbstractDao abstract class serves as a generic base class for data access classes. It provides the most common operations to retrieve, update and delete entities as well as to build HQL and native SQL queries.

It is possible to add custom DAOs to the platform. To accomplish this one needs to create a DAO interface extending either ReadOnlyDao or ReadWriteDao, add custom operations such as entity specific finders and then create a custom DAO class extending AbstractDao and implementing the custom DAO interface.

```java
public class MyEntity implements BaseEntityI {
    private long id;
    private String name;

    @Override
    public long getId() {
        return this.id;
    }

    protected void setId(final long id) {
        this.id = id;
    }

    @Override
    public boolean isInitialized() {
        return true;
    }

    public String getName() {
        return this.name;
    }

    public void setName(final String name) {
        this.name = name;
    }
}
```

```java
public interface MyEntityDao extends ReadWriteDao<MyEntity> {
    public MyEntity findByName(String name);
}
```
@Repository
public class MyEntityDaoImpl extends AbstractDao<MyEntity> implements MyEntityDao {

    public MyEntityDaoImpl(final SessionFactory sessionFactory) {
        super(MyEntity.class, sessionFactory);
    }

    @Override
    public Strategy findByName(final String name) {
        return findUniqueCaching("from MyEntity where name = :name", QueryType.HQL, new NamedParam("name", name));
    }
}

HQL and SQL queries used by AlgoTrader DAO components are externalized and stored in Hibernate.hbm.xml file. This allows for better management and for easier re-use of queries.

<query name='Strategy.findByName'>
  <![[CDATA[
    from StrategyImpl
    where name = :name
  ]]>}
</query>

Queries can be accessed from DAO classes or custom components by their names

public class StrategyDaoImpl extends AbstractDao<Strategy> implements StrategyDao {
    ...
    @Override
    public Strategy findByName(final String name) {
        return findUniqueCaching("Strategy.findByName", QueryType.BY_NAME, new NamedParam("name", name));
    }
}
3.3. Services

3.3.1. Private Services

Table 3.1. Private Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlgoOrderService</td>
<td>Order Services responsible for handling of AlgoOrders (delegates to AlgoOrderExecServices)</td>
</tr>
<tr>
<td>AlgoOrderExecService</td>
<td>Abstract Base Class for all Algo Execution Order Services</td>
</tr>
<tr>
<td>ExternalAccountService</td>
<td>Abstract Base Class for all external Account Interfaces</td>
</tr>
<tr>
<td>ExternalMarketDataService</td>
<td>Abstract Base Class for all external Market Data Interfaces</td>
</tr>
<tr>
<td>FixSessionService</td>
<td>Exposes properties of FIX sessions</td>
</tr>
<tr>
<td>ForexService</td>
<td>Responsible for the FX Hedging functionality</td>
</tr>
<tr>
<td>GenericOrderService</td>
<td>Parent Class for all Order Services</td>
</tr>
<tr>
<td>MarketDataPersistenceService</td>
<td>Responsible for persisting Market Data to the database</td>
</tr>
<tr>
<td>OrderExecutionService</td>
<td>Responsible for handling of persistence and propagation various trading events such as order status update and order fills as well as maintaining order execution status in the order book.</td>
</tr>
<tr>
<td>OrderPersistenceService</td>
<td>Responsible for persisting Orders and OrderStatus to the database</td>
</tr>
<tr>
<td>ReconciliationService</td>
<td>Responsible for reconciliation of reports provided by the broker</td>
</tr>
<tr>
<td>ResetService</td>
<td>Responsible for resetting the DB state (e.g. before the start of a simulation)</td>
</tr>
<tr>
<td>ServerLookupService</td>
<td>Provides internal data lookup operations to other server side services</td>
</tr>
<tr>
<td>SimpleOrderService</td>
<td>Order Service responsible for handling of Simple Orders (delegates to SimpleOrderExecServices)</td>
</tr>
<tr>
<td>SimpleOrderExecService</td>
<td>Abstract Base Class for all Simple Order Execution Services</td>
</tr>
<tr>
<td>StrategyPersistenceService</td>
<td>Handles persistence of Strategy Entities</td>
</tr>
<tr>
<td>TransactionPersistenceService</td>
<td>Responsible for the persistence of Transactions, Positions updates and Cash Balance updates to the database</td>
</tr>
<tr>
<td>TransactionService</td>
<td>Responsible for handling of incoming Fills</td>
</tr>
</tbody>
</table>

3.3.2. Order Services

All OrderServices are derived from the GenericOrderService. All Order Services based on Fix are derived from FixOrderService which in terms has subclasses for Fix versions 4.2 and 4.4.
3.3.3. Market Data Services

All MarketDataServices are derived from the general ExternalMarketDataService. All MarketDataServices based on Fix are derived from FixMarketDataService which in terms has subclasses for Fix versions 4.2 and 4.4.

3.3.4. Historical Data Services

Historical Data Services are used to download aggregated Market Data Events from the Market Data Provider.

3.3.5. Reference Data Services

Reference Data Services are used to download current option and future chains as well as information about stocks.
Java Environment

4.1. AlgoTrader Project Structure

The Framework AlgoTrader consists of the following Sub-Projects:

algotrader
   the main project

algotrader-common
   contains java code accessible to trading strategies

algotrader-core
   contains internal java code needed by the AlgoTrader Server

algotrader-dm
   contains the java code for the AlgoTrader reference and historical data management UIs

bootstrap/algotrader-conf
   contains the configuration files needed by the AlgoTrader Server

bootstrap/algotrader-launch
   contains the Run Configurations

algotrader-archetype-esper
   Maven archetype for creating new AlgoTrader Esper based strategies

algotrader-archetype-java
   Maven archetype for creating new AlgoTrader Java-only strategies

algotrader-archetype-simple
   Maven archetype for creating new AlgoTrader simple strategies

at-tests/algotrader-test-framework
   test framework for automated end-to-end UI tests

at-tests/algotrader-ui-tests
   automated UI tests

at-tests/algotrader-application-tests
   automated tests covering internal and REST-based functionalities

at-tests/algotrader-test
   contains integration tests based on JUnit ¹

4.1.1. common project

the AlgoTrader common project has the following structure:

¹ https://junit.org/
4.1.2. core project

the AlgoTrader core project has the following structure:

Table 4.2. core project

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/main/java</td>
<td>manually created class files</td>
</tr>
<tr>
<td>src/main/resources</td>
<td>manually created configuration files</td>
</tr>
<tr>
<td>target/generated-sources/</td>
<td>generated class files</td>
</tr>
<tr>
<td>main/java</td>
<td></td>
</tr>
<tr>
<td>src/test/java</td>
<td>JUnit test cases</td>
</tr>
<tr>
<td>src/test/resources</td>
<td>test resources</td>
</tr>
</tbody>
</table>

4.1.3. conf project

the AlgoTrader conf project has the following structure:

Table 4.3. conf project

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/main/resources</td>
<td>all the properties files</td>
</tr>
</tbody>
</table>

4.1.4. launch project

the AlgoTrader conf project has the following structure:

Table 4.4. launch project

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>all the Run Configuration files</td>
</tr>
</tbody>
</table>
4.1.5. strategy projects

Strategy projects typically have the following structure:

Table 4.5. strategy projects

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/main/java</td>
<td>java code</td>
</tr>
<tr>
<td>src/main/resources</td>
<td>config files</td>
</tr>
<tr>
<td>bin</td>
<td>Run Configuration and shell start scripts</td>
</tr>
<tr>
<td>log</td>
<td>log files</td>
</tr>
</tbody>
</table>

4.2. Java Packages & Classes

For a full list of java packages and classes please visit our Javadoc²

4.3. Maven Environment

AlgoTrader uses Maven³ as its build management framework. Every project/module therefore has its own pom.xml (Project Object Model) defining its structure and dependencies.

Figure 4.1. Maven Dependencies

² http://doc.algotrader.ch/javadoc/index.html
³ http://maven.apache.org/
4.3.1. Maven assemblies

AlgoTrader provides maven assemblies for both the AlgoTrader core module as well as for strategies. The core assembly declares all components required to run AlgoTrader server in distributed mode.

The server maven assembly definition file is located in /algotrader-core/src/main/assembly/server-bin.xml

The following command generates binary deployment packages from assembly descriptors.

mvn clean package

Binary deployment packages generated from Maven assemblies come in two varieties: tar.gz and zip, the former being more optimized for Unix-like operating systems while the latter being suitable for Windows platforms.

4.3.2. Packaging strategies

In addition to the AlgoTrader server assembly there is also an assembly file available that can be used to package strategies.

The strategy maven assembly definition file is located in /algotrader-core/src/main/assembly/strategy-bin.xml. To use the strategy assembly in your trading strategy add the following plugin to the strategy pom.xml:

```xml
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.5</version>
  <dependencies>
    <dependency>
      <groupId>algotrader</groupId>
      <artifactId>algotrader-core</artifactId>
      <version>${algotrader.version}</version>
    </dependency>
  </dependencies>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <descriptorRefs>
          <descriptorRef>strategy-bin</descriptorRef>
        </descriptorRefs>
        <tarLongFileMode>gnu</tarLongFileMode>
      </configuration>
    </execution>
  </executions>
</plugin>
```
Chapter 5.  

Code Generation

Java Entities, Entity Interfaces and Value Objects are created by the means of the Hibernate Tools project\(^1\) using the hbm2java code exporter\(^2\).

The Hibernate Tools project provides the code generator as an Eclipse plugin as well as a set of Ant tasks. AlgoTrader provides a custom maven plugin called maven-codegen-plugin which wraps the code generator.

```
<groupId>algotrader</groupId>
<artifactId>model-codegen-plugin</artifactId>
<name>Model code generator plugin</name>
<version>0.1.5</version>
```

The maven-codegen-plugin has been added to the file /algotrader/common/pom.xml

```
<plugin>
  <groupId>algotrader</groupId>
  <artifactId>model-codegen-plugin</artifactId>
  <version>0.1.5</version>
  <executions>
    <execution>
      <id>generate-model</id>
      <goals>
        <goal>codegen</goal>
      </goals>
      <configuration>
        <templates>
          <template>
            <file>pojo/Pojo.ftl</file>
            <pattern>{package-name}/{class-name}.java</pattern>
          </template>
          <template>
            <file>pojo/Interface.ftl</file>
            <pattern>{package-name}/{class-name}I.java</pattern>
          </template>
          <template>
            <file>pojo/ValueObject.ftl</file>
            <pattern>{package-name}/{class-name}VO.java</pattern>
          </template>
          <template>
            <file>pojo/ValueObjectBuilder.ftl</file>
            <pattern>{package-name}/{class-name}VOBuilder.java</pattern>
          </template>
        </templates>
      </configuration>
    </execution>
  </executions>
</plugin>
```

\(^1\) [http://hibernate.org/tools/](http://hibernate.org/tools/)
The above configuration generates the following artifacts for each Java Entity defined in the Hibernate mapping files:

- Entity
- Entity interface
- Value Object
- Value Object Builder

The code generator uses Hibernate mapping files which are located in /algotrader/common/src/main/resources in combination with Freemarker\(^3\) templates which are located in /algotrader/common/pojo. These templates are based on the original version supplied with the Hibernate Tools project but have been augmented to produces Java code needed by AlgoTrader. For this purpose several custom attributes have been added to Hibernate mapping files:

**Table 5.1. Hibernate mapping files - custom meta attributes**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>implements</td>
<td>The interface this Entity implements</td>
</tr>
<tr>
<td>generated-class</td>
<td>Name of the main Entity class file</td>
</tr>
<tr>
<td>class-code</td>
<td>Extra class code generated into Entity classes</td>
</tr>
<tr>
<td>interface-code</td>
<td>Extra class code generated into Entity interfaces</td>
</tr>
<tr>
<td>vo-code</td>
<td>Extra class code generated into Value Objects</td>
</tr>
<tr>
<td>class-description</td>
<td>Javadoc comment for classes</td>
</tr>
<tr>
<td>field-description</td>
<td>Javadoc comment for fields</td>
</tr>
<tr>
<td>use-in-equals</td>
<td>should this property be used in the equals method</td>
</tr>
<tr>
<td>property-type</td>
<td>the class to be used for this property</td>
</tr>
</tbody>
</table>

Generated code is placed under the directory /algotrader/common//target/generated-sources/main/java.

---

\(^3\) https://freemarker.apache.org/
Database

6.1. Database scripts

- algotrader/core/src/main/resources/db/h2 H2 scripts loaded at runtime
  - h2.sql h2 database schema file
- algotrader/core/src/main/scripts/db/mysql MySql scripts
  - create-fix-messagestore.sql creates database tables uses for Fix message persistence
  - datetime_milli_precision.sql converts all datetime fields to usage of millisecond (requires MySql 5.6.4)
  - mysql-to-h2.sql batch script to convert MySQL schema to h2 schema.
  - mysql-to-h2-data.sql batch script to convert MySQL data to h2 schema.
  - reset-db.sql reset script which resets the tables cash_balance, order, portfolio_value, subscription, transaction and position.

6.2. Transaction Handling

Using Spring Transaction Boundaries are declared on Services using the annotation @Transactional. Transaction Boundaries are handled by the org.springframework.transaction.interceptor.TransactionInterceptor. A typical declaration looks like this:

```java
@Transactional(propagation = Propagation.SUPPORTS)
public class TransactionServiceImpl implements TransactionService {

    @Transactional(propagation = Propagation.REQUIRED)
    public void saveTransaction(final Transaction transaction) {
        ...
    }
    ...
}
```

In order for transactions to work services need to implement an interface (e.g. TransactionServiceImpl implements TransactionService).
Note

Transaction demarcation only works when a service method is called from another service with dependencies properly injected through Spring. For example using a scheduler inside a service to then call methods with itself will not create transactions.
Market Data

All Market Data Interfaces have a set of unique artifacts:

- **Configuration Files** (`conf-ib.properties` and `conf-bb.properties`)
- **Adapter Classes** (e.g. `IBAdapter`) responsible for management of Sessions. Adapters are available over JMX
- **Session Classes** (e.g. `IBSession`). Representing an individual Market Data Session
- **Message Handler Classes** (e.g. `BBMessageHandler`) responsible for receiving `MarketDataEvents` and propagating them into Esper
- **Esper Modules** (e.g. `market-data-ib`) responsible for processing and filtering of `MarketDataEvents`

Processing of Market Data is handled through the `MarketDataService`, which calls the market data provider specific `ExternalMarketDataService` implementations. Every market data service has to provide implementation of this interface e.g. (e.g. `IBMarketDataServiceImpl` or `BBMarketDataServiceImpl`).

The most important methods provided by the `MarketDataService` are `subscribe` and `unsubscribe`. Through the use of these methods new Market Data can be subscribed and unsubscribed. Subscribed securities are persisted within the DB-table `subscription`. The actual subscription of securities to the external broker is done through the market data provider specific `MarketDataService`.

Market data provider interfaces are responsible for receiving market data provider specific Market Data and sending them into the Esper Service Instance of the AlgoTrader Server. The Esper Service Instance will then convert these Events into generic `MarketDataEvents` (i.e. Ticks or Bars) which will be propagated to subscribed Strategies.
Adapters

8.1. Bloomberg

The Bloomberg infrastructure consists of the following classes:

Table 8.1. Bloomberg Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBAdapter</td>
<td>Management Adapter for the Bloomberg environment. Allows to start and stop Bloomberg Sessions</td>
</tr>
<tr>
<td>BBConstants</td>
<td>Bloomberg Constants used by the Bloomberg interface</td>
</tr>
<tr>
<td>BBIdGenerator</td>
<td>Generator for Bloomberg Request Id's</td>
</tr>
<tr>
<td>BBMessageHandler</td>
<td>Message Handler for incoming Bloomberg messages. Messages are either propagated into the Esper Engine or delegated back to the corresponding Service.</td>
</tr>
<tr>
<td>BBMarketDataMessageHandler</td>
<td></td>
</tr>
<tr>
<td>BBSession</td>
<td>Represents a connection to the Bloomberg service</td>
</tr>
</tbody>
</table>

8.2. Currenex

The Currenex infrastructure consists of the following classes:

Table 8.2. Currenex Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNXFixMarketDataServiceImpl</td>
<td>MarketData service implementation for the Currenex environment.</td>
</tr>
<tr>
<td>CNXFixOrderServiceImpl</td>
<td>Currenex specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>

8.3. DukasCopy

The DukasCopy infrastructure consists of the following classes:

Table 8.3. DukasCopy Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCFixMarketDataServiceImpl</td>
<td>MarketData service implementation for the DukasCopy environment.</td>
</tr>
<tr>
<td>DCFixOrderServiceImpl</td>
<td>DukasCopy specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>
8.4. Exante

The Exante infrastructure consists of the following classes:

Table 8.4. Exante Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XNTFixMarketDataServiceImpl</td>
<td>MarketData service implementation for the Exante environment.</td>
</tr>
<tr>
<td>XNTFixOrderServiceImpl</td>
<td>Exante specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>

8.5. EzeSoft/RealTick

The EzeSoft/RealTick infrastructure consists of the following classes:

Table 8.5. EzeSoft/RealTick Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTFixOrderServiceImpl</td>
<td>EzeSoft/RealTick specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>

8.6. Fix Interface

The Fix infrastructure consists of the following classes:

Table 8.6. Fix Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session</td>
<td>A Session represents a connection to a broker / exchange / market data provider</td>
</tr>
<tr>
<td>Application</td>
<td>For each Session an Application object is created. It will forward incoming messages to the corresponding MessageHandlers</td>
</tr>
<tr>
<td>DefaultFixApplication</td>
<td></td>
</tr>
<tr>
<td>FixApplicationFactory</td>
<td>Is responsible for the creation of Applications</td>
</tr>
<tr>
<td>DefaultFixApplicationFactory</td>
<td></td>
</tr>
<tr>
<td>FixMultiApplicationSessionFactory</td>
<td>Creates a Session and Application using the specified FixApplicationFactory according to the following steps:</td>
</tr>
<tr>
<td></td>
<td>• lookup the FixApplicationFactory by its name</td>
</tr>
<tr>
<td></td>
<td>• create an Application</td>
</tr>
</tbody>
</table>
## Class / Interface | Description
--- | ---
create a **DefaultSessionFactory** | **ExternalSessionStateHolder** Represents the current state of a Session (i.e. DISCONNECTED, CONNECTED, LOGGED_ON and SUBSCRIBED)
create a **Session** | **DefaultFixSessionStateHolder** A **ExternalSessionStateHolder** for market data sessions that will subscribe to securities as soon as the session is logged on.
**MarketDataFixSessionStateHolder** | Generator for Fix Order Ids. The default implementation reads the last Order Ids from the Fix log on start-up.
**FixOrderIdGenerator** | Management Adapter for the Fix environment. Allows the creation of dynamic sessions, sending Messages and managing Order Ids.
**DefaultFixOrderIdGenerator** | **ManagedFixAdapter** Manageable implementation of a **FixAdapter** (based on JMX)
**FixAdapter** | **EventPattern** QuickFix/J currently supports daily sessions (with a daily session 7 times a week) and weekly sessions (with one weekly session). However some brokers (e.g. JP Morgan) use daily sessions during workdays. To accomplish this scenario, AlgoTrader allows creation of a weekly **logon/logoff** event (e.g. Mo 08:00:00 and Fr 18:00:00) using Esper Statements
**FixEventScheduler** | **DefaultFixEventScheduler** A Spring Factory Bean that creates the **SocketInitiator** necessary for all Fix Sessions.
**FixSocketInitiatorFactoryBean** | **Fix42MarketDataMessageHandler** Message Handler for incoming Fix market data messages. These classes need to be sub classed by the corresponding market data interface. Messages are propagated into the Esper Engine.
**Fix42OrderMessageHandler** | **Fix44OrderMessageHandler** Message Handler for incoming Fix Execution Reports. These classes need to be sub classed by the corresponding order interface. Messages are propagated into the Esper Engine.
**Fix44MarketDataMessageHandler** |

### 8.7. Fortex

The Fortex infrastructure consists of the following classes:

#### Table 8.7. Fortex Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FTXFixMarketDataServiceImpl</strong></td>
<td>MarketData service implementation for the Fortex environment.</td>
</tr>
</tbody>
</table>
### 8.8. FXCM

The FXCM infrastructure consists of the following classes:

**Table 8.8. FXCM Infrastructure**

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FXCMFixMarketDataServiceImpl</td>
<td>MarketData service implementation for the FXCM environment.</td>
</tr>
<tr>
<td>FXCMFixOrderServiceImpl</td>
<td>FXCM specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>

### 8.9. IB Native Interface

The IB infrastructure consists of the following classes:

**Table 8.9. IB Infrastructure**

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBSession</td>
<td>An IBSession represents a connection to the TWS or IB Gateway</td>
</tr>
<tr>
<td>IBSessionStateHolder</td>
<td>Represents the current state of a Session (i.e. DISCONNECTED, CONNECTED, IDLE, LOGGED_ON and SUBSCRIBED)</td>
</tr>
<tr>
<td>DefaultIBSessionStateHolder</td>
<td>Represents the current state of a Session (i.e. DISCONNECTED, CONNECTED, IDLE, LOGGED_ON and SUBSCRIBED)</td>
</tr>
<tr>
<td>IBAdapter</td>
<td>Management Adapter for the IB environment. Allows to connect and disconnect IB Sessions as well as retrieval of the current ConnectionState</td>
</tr>
<tr>
<td>DefaultIBAdapter</td>
<td>Management Adapter for the IB environment. Allows to connect and disconnect IB Sessions as well as retrieval of the current ConnectionState</td>
</tr>
<tr>
<td>AbstractIBMessageHandler</td>
<td>Message Handler for incoming IB messages. Messages are either propagated into the Esper Engine or delegated back to the corresponding Service.</td>
</tr>
<tr>
<td>DefaultIBMessageHandler</td>
<td>Message Handler for incoming IB messages. Messages are either propagated into the Esper Engine or delegated back to the corresponding Service.</td>
</tr>
<tr>
<td>IBOrderMessageFactory</td>
<td>Factory for order messages.</td>
</tr>
<tr>
<td>DefaultIBOrderMessageFactory</td>
<td>Factory for order messages.</td>
</tr>
<tr>
<td>IBExection</td>
<td>Details of an individual order execution</td>
</tr>
<tr>
<td>IBExections</td>
<td>Collection (internally represented by a map) of all order executions</td>
</tr>
<tr>
<td>IBPendingRequest</td>
<td>Details of an individual data request such as historic data or contract details</td>
</tr>
</tbody>
</table>
8.10. JP Morgan

The JP Morgan infrastructure consists of the following classes:

**Table 8.10. JP Morgan Infrastructure**

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBPendingRequests</td>
<td>Collection (internally represented by a map) of all pending data requests</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPMFixOrderServiceImpl</td>
<td>JP Morgan specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>

8.11. LMAX

The LMAX infrastructure consists of the following classes:

**Table 8.11. LMAX Infrastructure**

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMAXFixMarketDataServiceImpl</td>
<td>MarketData service implementation for the LMAX environment.</td>
</tr>
<tr>
<td>LMAXFixOrderServiceImpl</td>
<td>LMAX specific implementation of the generic FIX adapter for order management.</td>
</tr>
<tr>
<td>LMAXDropCopyAllocationServiceImpl</td>
<td>Allocation of external fills to the right internal security and account.</td>
</tr>
</tbody>
</table>

8.12. Nexus Prime

The Nexus Prime infrastructure consists of the following classes:

**Table 8.12. Nexus Prime Infrastructure**

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NXSFixMarketDataServiceImpl</td>
<td>MarketData service implementation for the Nexus Prime environment.</td>
</tr>
<tr>
<td>NXSFixOrderServiceImpl</td>
<td>Nexus Prime specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>

8.13. PrimeXM

The PrimeXM infrastructure consists of the following classes:
Table 8.13. PrimeXM Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PXMFixMarketDataServiceImpl</td>
<td>MarketData service implementation for the PrimeXM environment.</td>
</tr>
<tr>
<td>PXMFixOrderServiceImpl</td>
<td>PrimeXM specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>

8.14. Quandl

The Quandl infrastructure consists of the following classes:

Table 8.14. Quandl Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QDLHistoricalDataServiceImpl</td>
<td>Historical data service implementation for the Quandl environment.</td>
</tr>
</tbody>
</table>

8.15. QuantHouse

The QuantHouse infrastructure consists of the following classes:

Table 8.15. QuantHouse Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QHAdapter</td>
<td>Management Adapter for the QuantHouse environment.</td>
</tr>
<tr>
<td>QHMessageHandler</td>
<td>Message Handler for incoming QuantHouse messages. Messages are either propagated into the Esper Engine or delegated back to the corresponding Service.</td>
</tr>
<tr>
<td>QHSessionStateHolder</td>
<td>Holds the current state of the QuantHouse session</td>
</tr>
</tbody>
</table>

8.16. SocGen

The SocGen infrastructure consists of the following classes:

Table 8.16. SocGen Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGFixOrderServiceImpl</td>
<td>SocGen specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>

8.17. Trading Technologies

The Trading Technologies infrastructure consists of the following classes:
Table 8.17. Trading Technologies Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTFixMarketDataServiceImpl</td>
<td>MarketData service implementation for the Trading Technologies environment.</td>
</tr>
<tr>
<td>TTFixOrderServiceImpl</td>
<td>Trading Technologies specific implementation of the generic FIX adapter for order management.</td>
</tr>
<tr>
<td>TTDropCopyAllocationServiceImpl</td>
<td>Allocation of external fills to the right internal security and account.</td>
</tr>
<tr>
<td>TTFixReferenceDataServiceImpl</td>
<td>Service to acquire the list of available securities from Trading Technologies.</td>
</tr>
</tbody>
</table>

8.18. UBS

The UBS infrastructure consists of the following classes:

Table 8.18. UBS Infrastructure

<table>
<thead>
<tr>
<th>Class / Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UBSFixOrderService</td>
<td>MarketData service implementation for the UBS environment.</td>
</tr>
<tr>
<td>UBSFixOrderServiceImpl</td>
<td>UBS specific implementation of the generic FIX adapter for order management.</td>
</tr>
</tbody>
</table>


Table 8.19. Main service classes

<table>
<thead>
<tr>
<th>Service class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNCOrderServiceImpl</td>
<td>Order submission</td>
</tr>
<tr>
<td>BNCMarketDataServiceImpl</td>
<td>Market data feed</td>
</tr>
<tr>
<td>BNCReferenceDataServiceImpl</td>
<td>Reference data - instruments</td>
</tr>
<tr>
<td>BNCAccountServiceImpl</td>
<td>Account info</td>
</tr>
</tbody>
</table>

Table 8.20. Main classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNCAdapter</td>
<td>Main adapter class</td>
</tr>
<tr>
<td>BNCOrderMessageHandler,</td>
<td>Live market data subscription connectors</td>
</tr>
<tr>
<td>BNCMarketDataMessageHandler</td>
<td></td>
</tr>
<tr>
<td>BNCServiceWiring, BNCWiring</td>
<td>Spring config files</td>
</tr>
</tbody>
</table>
8.20. Bitfinex

Table 8.21. Main service classes

<table>
<thead>
<tr>
<th>Service class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFXOrderServiceImpl</td>
<td>Order submission</td>
</tr>
<tr>
<td>BFXMarketDataServiceImpl</td>
<td>Market data feed</td>
</tr>
<tr>
<td>BFXReferenceDataServiceImpl</td>
<td>Reference data - instruments</td>
</tr>
<tr>
<td>BFXAccountServiceServiceImpl</td>
<td>Account info retrieval</td>
</tr>
</tbody>
</table>

Table 8.22. Main classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFXRestAdapter</td>
<td>Main adapter class</td>
</tr>
<tr>
<td>BFXWebSocketAdapter, BFXMessageHandler</td>
<td>WebSockets connectivity logic</td>
</tr>
<tr>
<td>BFXServiceWiring, BFXWiring</td>
<td>Spring config files</td>
</tr>
</tbody>
</table>

8.21. Bitflyer

Table 8.23. Main service classes

<table>
<thead>
<tr>
<th>Service class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFLOrderServiceImpl</td>
<td>Order submission</td>
</tr>
<tr>
<td>BFLMarketDataServiceImpl</td>
<td>Market data feed</td>
</tr>
<tr>
<td>BFLReferenceDataServiceImpl</td>
<td>Reference data - instruments</td>
</tr>
<tr>
<td>BFLAccountServiceServiceImpl</td>
<td>Account info</td>
</tr>
</tbody>
</table>

Table 8.24. Main classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFLRestAdapter</td>
<td>Main adapter class</td>
</tr>
<tr>
<td>BFLPubNubAdapter, BFLMessageHandler</td>
<td>Live market data subscription connectors</td>
</tr>
<tr>
<td>BFLServiceWiring, BFLWiring</td>
<td>Spring config files</td>
</tr>
</tbody>
</table>

8.22. BitMEX

Table 8.25. Main service classes

<table>
<thead>
<tr>
<th>Service class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMXOrderServiceImpl</td>
<td>Order submission: Market, Limit, Stop and Stop-Limit orders with DAY, GTC, FOK or IOC Time in Force</td>
</tr>
</tbody>
</table>
### Service class name | Functionality
---|---
BMXMarketDataService_Impl | Market data feed
BMXReferenceDataService_Impl | Reference data - instruments
BMXAccountService_Impl | Account balances and info retrieval

**Table 8.26. Main classes**

<table>
<thead>
<tr>
<th>Class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMXRestAdapter</td>
<td>Main adapter class</td>
</tr>
<tr>
<td>BMXWebSocketAdapter, BMXMarketDataMessageHandler, BMXOrderMessageHandler</td>
<td>Live market data subscription and order status update connectors</td>
</tr>
<tr>
<td>BMXServiceWiring, BMXWiring</td>
<td>Spring config files</td>
</tr>
</tbody>
</table>

### Bitstamp

**Table 8.27. Main service classes**

<table>
<thead>
<tr>
<th>Service class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTSFixOrderServiceImpl</td>
<td>Order submission</td>
</tr>
<tr>
<td>BTSFixMarketDataService_Impl</td>
<td>Market data feed</td>
</tr>
<tr>
<td>BTSReferenceDataService_Impl</td>
<td>Reference data - instruments</td>
</tr>
<tr>
<td>BTSAccountServiceImpl</td>
<td>Account balances and info retrieval</td>
</tr>
</tbody>
</table>

**Table 8.28. Main classes**

<table>
<thead>
<tr>
<th>Class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTSRestAdapter</td>
<td>Main adapter class</td>
</tr>
<tr>
<td>BTSFixMessageHandler</td>
<td>Facade class for handling Fix messages</td>
</tr>
<tr>
<td>BTSServiceWiring, BTSWiring, BTSFixServiceWiring, BTSFixWiring</td>
<td>Spring config files</td>
</tr>
</tbody>
</table>

### CoinAPI

**Table 8.29. Main service classes**

<table>
<thead>
<tr>
<th>Service class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNPHistoricalDataService_Impl</td>
<td>Historical data</td>
</tr>
<tr>
<td>CNPMarketDataService_Impl</td>
<td>Market data feed</td>
</tr>
<tr>
<td>CNPReferenceDataService_Impl</td>
<td>Reference data - instruments</td>
</tr>
</tbody>
</table>
Table 8.30. Main classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNPRestAdapter</td>
<td>Main adapter class</td>
</tr>
<tr>
<td>CNPWebSocketAdapter, CNPMessageHandler</td>
<td>WebSockets connectivity logic</td>
</tr>
<tr>
<td>CNPServiceWiring, CNPWiring</td>
<td>Spring config files</td>
</tr>
</tbody>
</table>

8.25. Coinbase

Table 8.31. Main service classes

<table>
<thead>
<tr>
<th>Service class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNBMarketDataServiceImpl</td>
<td>MarketData service implementation for the Coinbase environment.</td>
</tr>
<tr>
<td>CNBFixOrderServiceImpl</td>
<td>Coinbase specific implementation of the generic FIX adapter for order management.</td>
</tr>
<tr>
<td>CNBAccountServiceImpl</td>
<td>Account specific functionalities offered by the Coinbase API.</td>
</tr>
<tr>
<td>CNBReferenceDataServiceImpl</td>
<td>Service to acquire the list of available securities from Coinbase.</td>
</tr>
</tbody>
</table>

8.26. Coinigy

Table 8.32. Main service classes

<table>
<thead>
<tr>
<th>Service class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNGOrderServiceImpl</td>
<td>Order submission</td>
</tr>
<tr>
<td>CNGMarketDataServiceImpl</td>
<td>Market data feed</td>
</tr>
<tr>
<td>CNGReferenceDataServiceImpl</td>
<td>Reference data - instruments</td>
</tr>
<tr>
<td>CNGAccountServiceImpl</td>
<td>Account info</td>
</tr>
</tbody>
</table>

Table 8.33. Main classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNGRestAdapter</td>
<td>A REST client for the Coinigy REST API endpoint. Allows placing Limit and Stop Limit orders as well as retrieval of reference and account data.</td>
</tr>
<tr>
<td>CNGSocketClusterAdapter,CNGMarketDataMessageHandler</td>
<td>Socket Cluster client for the Coinigy WebSocket API endpoint that provides real-time market data feeds. Message Handler for incoming market data updates received through the WebSocket channels.</td>
</tr>
</tbody>
</table>
8.27. CoinMarketCap

Table 8.34. Main service classes

<table>
<thead>
<tr>
<th>Service class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMCHistoricalDataServiceImpl</td>
<td>Daily historical data</td>
</tr>
<tr>
<td>CMCReferenceDataServiceImpl</td>
<td>Reference data - instruments</td>
</tr>
</tbody>
</table>

Table 8.35. Main classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMCRestAdapter</td>
<td>Main adapter class</td>
</tr>
<tr>
<td>CMCWiring, CMCServiceWiring</td>
<td>Spring config files</td>
</tr>
</tbody>
</table>
Execution Algos

9.1. Development of Execution Algos

Additional Execution Algos can be added to the system with relatively minimal effort. Execution Algos consist of the following artifacts

- a subclass of AlgoOrder. An AlgoOrder should be a plain old Java bean and contain no execution logic.

  Note
  AlgoOrder subclasses are persistent via hibernate, like other Order entities. Adding a new AlgoOrder subclass requires registering it in the hibernate mapping file: Order.hbm.xml. All hibernate constraints which apply to Order will apply also to AlgoOrder subclass. Please mind these constraints, esp. mandatory fields (e.g. security) while developing the algo logic. If any constraint is violated the algo order persistence will fail.

- an object that represents the state of an algo order execution which needs to subclass AlgoOrderStateVO. Please note that if state objects contains elements that are potentially threading unsafe, access to those elements must be synchronized!

- an esper module (optional). This module can optionally provide statements to cancel a child order, modify a child order, send the next child order, etc. The Esper module has to be registered in the conf-core.properties file by modifying the server-engine.init property

- an implementation of AlgoOrderExecService interface. It is generally recommended to subclass AbstractAlgoOrderExecService and implement its protected methods that represent various algo specific handling logic. The #handleValidateOrder method must implement algo specific order validation logic. The #handleSendOrder, #handleModifyOrder and #handleCancelOrder method must implement algo specific order execution, modification and cancellation logic respectively.

- a corresponding entry in the ch.algotrader.enumeration.OrderType class.

- a corresponding entry in the order_preference MySQL table. The new algo order type has to be added to the database ORDER_TYPE enum.

- a Spring wiring within ch.algotrader.wiring.core.ServiceWiring.

Custom AbstractAlgoOrderExecService implementation can also tap into event streams pertaining to the algo order execution. The #handleChildFill and #handleChildOrderStatus methods can be used to provide custom handling of fills and status events for child orders executed by the algo and to update its internal state. The #handleOrderStatus method can be used to update the internal state in response to transition of the algo order from one execution phase to another

In addition the class can implement any additional logic needed in conjunction with custom Esper statements.
Note

It is important that AlgoOrderExecService implementations are built to be state-less. They must store all details pertaining to execution of algo orders in their respective state objects.

- Register the new algo in MetaDataRestController That will allow you to display the order properties in the UI's grids and to place that order through the UI's Advanced Order Form (the form will generate automatically, see XXXX)

- If one's going to place Execution Algos from UI, one should also provide a AlgoOrderVOMixIn mapper class and register it to ObjectMapperFactory. Also a conversion from VO to order object hast to be added to OrderServiceImpl in this case

The OrderService is aware of all AlgoOrderExecService instances declared in the Spring application context of the server process. Custom AlgoOrderExecService implementations also get automatically recognized as long as they are declared in the same application context. The OrderService delegates handling of individual orders to their respective algo service based on the order type. It is important for classes implementing AlgoOrderExecService to correctly implement its #getAlgoOrderType method.

9.2. Execution Algos entry form generation

The AlgoTrader UI can generate the entry form for Execution Algo automatically. To enable it one needs to register the AlgoOrderVO class in the MetaDataRestController and annotate it with @AlgoOrderMetaData. One needs to provide a mapping to the OrderType enum value there. These two steps will allow the UI to generate the entry form for that Execution Algo which will then be available in Advanced Order Form modal.

The Algo Order UI generation can be further tuned by annotating fields inside the AlgoOrderVO class with following annotations:

- ch.algotrader.entity.trade.algo.UIGeneration - allows to provide following properties:
  - description - adds additional information on UI for that field
  - name - allows to override the generated label for the field; by default the label is generated from the field's name in code
  - required - makes the field mandatory on UI
  - disabled - makes the field disabled on UI (but still visible)
  - hidden - hides the field on UI completely
  - order - allows to reorder fields. Without that annotation fields are displayed in the same order they're defined in the class. With this annotation one can provide any arbitrary order number and UI will sort all fields accordingly.
• **toggle** - allows to hide or disable field based on a boolean value of a field (see example below)

• **percentInput** - sets the field as a percent input, i.e. if the field is a decimal field with value ranging from 0 to 1 then it can be rendered on UI as a field with values from 0% to 100%

• **javax.validation.constraints.Pattern** - allows to define regex pattern for string values

• **javax.validation.constraints.Max** - allows to define max value for integers

• **javax.validation.constraints.Min** - allows to define min value for integers

• **javax.validation.constraints.DecimalMax** - allows to define max value for floating point numbers

• **javax.validation.constraints.DecimalMin** - allows to define min value for floating point numbers

Below code snipped shows an example for a `BigDecimal` field named `fieldA`. This field controls the visibility of `fieldB` and `fieldC`. `fieldB` is only visible if the value of `fieldA` is `true` and `fieldC` is only visible if the value of `fieldA` is `false`.

```java
@UIGeneration(toggle = @FieldToggle(visibleOnTrue = "fieldB", visibleOnFalse = "fieldC"))
private BigDecimal fieldA;

@UIGeneration()
private BigDecimal fieldB;

@UIGeneration()
private BigDecimal fieldC;
```
Spring Services

AlgoTrader is built on top of the Spring Framework, which uses BeanFactory and ApplicationContext to locate Spring Beans (= AlgoTrader-Services).

The *Spring*\(^1\) web site provides documentation\(^2\) such as *The IoC container*\(^3\) as an introduction.

AlgoTrader provides the class `ch.algotrader.ServiceLocator` which will instantiate the adequate BeanFactories & ApplicationContexts for a given operational mode depending on the specified `BEAN_REFERENCE_LOCATION`.

In Simulation mode the AlgoTrader Server as well as the Strategy run inside the same JVM.

In Live-Trading mode the AlgoTrader Server and strategies can be run in different JVMs. Through the use of `RmiServiceExporters` and `RmiProxyFactoryBean`, Strategies can call Services from the AlgoTrader Server. Behind the scenes this is handled transparently through RMI.

Please see *Remoting and web services using Spring*\(^4\) for further details.

10.1. Wiring Factories

AlgoTrader provides the following Wiring Factories, which are instantiated by the `ServiceLocators`:

<table>
<thead>
<tr>
<th>Wiring Factory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalWiringFactory</td>
<td>used when no remoting or strategy related functionality is needed (e.g. HistoricalDataStarter)</td>
</tr>
<tr>
<td>EmbeddedWiringFactory</td>
<td>used in Live Trading Mode when running in embedded mode</td>
</tr>
<tr>
<td>ServerWiringFactory</td>
<td>used in Live Trading Model on the server side</td>
</tr>
<tr>
<td>ClientWiringFactory</td>
<td>used by the Strategies in Live Trading Mode to connect to Services through RMI</td>
</tr>
<tr>
<td>SimulationWiringFactory</td>
<td>used in simulation</td>
</tr>
</tbody>
</table>

10.2. Application Context

AlgoTrader provides the following Wiring Classes and Application Context XML-Files:

---

1. https://spring.io
2. https://docs.spring.io/spring/docs/4.3.18.RELEASE/spring-framework-reference/htmlsingle/
3. https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans
Table 10.2. Application Context Files

<table>
<thead>
<tr>
<th>ApplicationContext</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommonWiring</td>
<td>contains common beans</td>
<td>Configuration, JMX Management, Esper Engine, Event Dispatching,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logging &amp; Web Configuration</td>
</tr>
<tr>
<td>ServerWiring</td>
<td>contains server bean definitions</td>
<td>ActiveMQ, SSL &amp; REST Controllers</td>
</tr>
<tr>
<td>CoreWiring</td>
<td></td>
<td>Caching, Configuration, DAOs, Data Source and Transaction Management,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hibernate, Live Cycle Manager, Server Esper Engine, Simulation &amp; InfluxDB</td>
</tr>
<tr>
<td>AdapterWiring</td>
<td>contains adapter related beans</td>
<td>market data and trading adapters</td>
</tr>
<tr>
<td>ExternalWiring</td>
<td>contains external services</td>
<td>market data and trading services</td>
</tr>
<tr>
<td>ClientServicesWiring</td>
<td>contains all client service beans</td>
<td>SubscriptionService, MarketDataCache, LiveCycleManager, CacheManager &amp; LookupService</td>
</tr>
<tr>
<td>EmbeddedJMSWiring</td>
<td>contains JMS embedded beans</td>
<td>Embedded JMS</td>
</tr>
<tr>
<td>Client.JMSWiring</td>
<td>contains JMS client beans</td>
<td>Remote JMS</td>
</tr>
<tr>
<td>JMSWiring</td>
<td>contains JMS server beans</td>
<td>Server JMS</td>
</tr>
<tr>
<td>applicationContext-export-remoteServices.xml</td>
<td>contains all RmiServiceExporters to make Services remotely available</td>
<td></td>
</tr>
<tr>
<td>applicationContext-import-remoteServices.xml</td>
<td>contains all RmiProxyFactoryBean to call remote Services from the Strategies through RMI</td>
<td></td>
</tr>
<tr>
<td>applicationContext-client-*.xml</td>
<td>to be provided by the Strategies</td>
<td>Strategy services, JMS queues</td>
</tr>
<tr>
<td>applicationContext-env.xml</td>
<td>contains environment specific</td>
<td>Mail Settings, Reconciliation Dispositions</td>
</tr>
<tr>
<td></td>
<td>bean definitions</td>
<td></td>
</tr>
</tbody>
</table>

The following table shows which Wiring Classes and ApplicationContext is referenced by which Wiring Factory:

Table 10.3. Application Context References

<table>
<thead>
<tr>
<th>ApplicationContext</th>
<th>Local</th>
<th>Embedded Server</th>
<th>Client</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommonWiring</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
### 10.3. Abstract Services

For many use cases abstract services are in place which can be extended for different broker interfaces.

For abstract services which have only one active implementation (through profiles), an alias can be defined for the concrete service (e.g. `iBHistoricalDataService`). A typical Spring Bean Alias definition looks like this:

```java
@Profile("bBHistoricalData")
@Bean(name = {"bBHistoricalDataService", "historicalDataService")
public HistoricalDataService createBBHistoricalDataService(
    final BBAdapter bBAdapter,
    final SecurityDao securityDao,
    final BarDao barDao) {
    return new BBHistoricalDataServiceImpl(bBAdapter, securityDao, barDao);
}
```

At runtime this service can now be accessed through its alias (e.g. `historicalDataService`)

For abstract services which might have more than one active implementation (through profiles), aliases are not available. In this case the following method can be used to look up all available concrete services that extend the abstract service (see `OrderService` for an example):

```java
ServiceLocator.instance().getServices(OrderExecService.class)
```
10.4. Service initialization order

`InitializingServiceI` interface represents an abstract service that requires special initialization after it has been created and fully wired in the Spring application context. The life-cycle manager automatically detects such services and calls their `InitializingServiceI#init()` method before proceeding with initialization of strategy engines and deployment of strategy modules. This helps ensure that all external interfaces are fully initialized prior to strategy activation. `InitializationPriority` annotation can be used to explicitly mark a service either as a part of the platform core or as a part of an external broker interface. Core services have higher priority than all other services and get initialized before all others.
Events and Messaging

AlgoTrader provides a sophisticated event dispatching and messaging sub system. In Simulation Mode as well as Embedded Mode Event Propagation takes places within the JVM. In Distributed Live Trading Mode Event Propagation from the AlgoTrader Server to the strategies (and between strategies) happens via JMS & ActiveMQ.

11.1. Embedded ActiveMQ message broker

AlgoTrader makes use of an embedded instance of ActiveMQ message broker for message dispatch and delivery. It presently supports three transports by default:

- VM: for internal message delivery
- TCP: for message delivery to strategies running in distributed mode
- WebSockets: for message delivery to the HTML5 front-end. WebSockets transport can also be used for message delivery to any arbitrary external application that supports WebSockets transport and STOMP messaging protocols.

11.2. Embedded Jetty HTTP server

In addition to RMI transport AlgoTrader provides a RESTful interface over HTTP/S. RESTful endpoints serve only a subset of AlgoTrader functionality primarily required for HTML5 front-end. While being a subset it nonetheless represents the core functionality of the platform.

HTTP/HTTPS transport is powered by the embedded Jetty HTTP server and REST endpoints are managed by Spring Web framework.

11.3. RESTful interface

RESTful endpoints largely expose the same interface as Spring services exposed via RMI. REST controllers must follow RESTful semantic and also use immutable value objects for input / output representation.

AlgoTrader RESTful controllers serve several purposes:

- they provide request / response mapping to RESTful endpoints and enforce a contract conforming with the principles of RESTful interface;
- they de-serialize endpoint input to immutable value objects and if necessary convert them to input structures expected by the services;
- they convert service output structures to immutable value objects that can be serialized by the endpoints;

1 http://activemq.apache.org/
they optionally perform additional input validation and mapping;
• they map service exception to endpoint responses with an appropriate error status;

The following RESTful controller provides a list of all accounts available in the system:

```java
@CrossOrigin
@RequestMapping(path = "account", method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
public List<AccountVO> getAccounts() {
    return lookupService.getAllAccounts().stream()
        .map(Account::convertToVO)
        .collect(Collectors.toList());
}
```

In this example the @CrossOrigin annotation marks endpoint as permitting cross origin requests. The @RequestMapping annotation defines various aspects of request / response mapping: path attribute defines path element of the request URI, method attribute defines the request method (such as GET, POST, PUT or DELETE), produces attribute defines expected media type of response body. The endpoint method implementation performs conversion of Account Entity objects to AccountVO objects, which are then serialized to JSON data stream by the framework.

For more detailed explanation of REST controllers and Web annotations please refer to Spring documentation.

AlgoTrader uses SWAGGER to document the individual REST endpoints, see:

### 11.4. Event Dispatcher

The EventDispatcher API represents a platform wide communication interface capable of submitting events to multiple Engine instances and event listeners both inside the same JVM as well as to separated JVMs. The EventDispatcher acts as an event bus for the AlgoTrader platform and individual strategies. The following Recipients are available

<table>
<thead>
<tr>
<th>Table 11.1. Event Recipients</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scope</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ALL</td>
</tr>
<tr>
<td>ALL_LOCAL</td>
</tr>
<tr>
<td>ALL_LOCAL_STRATEGIES</td>
</tr>
<tr>
<td>ALL_LOCAL_LISTENERS</td>
</tr>
<tr>
<td>ALL_STRATEGIES</td>
</tr>
</tbody>
</table>
11.5. Event Listeners

EventListener represents a generic communication interface to receive events from multiple event producers both in-process and remote. EventListenerRegistry interface represents a registry of event listeners used internally by the AlgoTrader Server process as well as individual strategy processes. One can register listeners for arbitrary event classes, which enables strategies to generate custom events either through Esper statements or in Java code and consume them internally or propagate them to other strategy processes.

The AlgoTrader platform provides a number of event listeners for common event types such as market data events, order events, external session events, life-cycle events, and a few others. Components that implement those event interfaces which are declared in the Spring application context get automatically registered with the platform upon initialization.

Table 11.2. Standard event listener classes

<table>
<thead>
<tr>
<th>EventListener</th>
<th>Receives</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarEventListener</td>
<td>BarVO events generated from tick events by individual strategies or fed from an external source</td>
</tr>
<tr>
<td>EntityCacheEventListener</td>
<td>EntityCacheEvictionEventVO generated by the cache manager</td>
</tr>
<tr>
<td>FillEventListener</td>
<td>FillVO events generated by trading interface adapters</td>
</tr>
<tr>
<td>GenericEventListener</td>
<td>GenericEventVO events generated by strategies</td>
</tr>
<tr>
<td>GenericTickEventListener</td>
<td>GenericTickVO events generated by market data interface adapters or fed from an external source</td>
</tr>
<tr>
<td>LifecycleEventListener</td>
<td>LifecycleEventVO generated by the life-cycle manager</td>
</tr>
<tr>
<td>OrderCompletionEventListener</td>
<td>OrderCompletionVO events generated by the Server Engine</td>
</tr>
<tr>
<td>OrderEventListener</td>
<td>OrderVO events generated by the order service</td>
</tr>
</tbody>
</table>
Event Listeners

<table>
<thead>
<tr>
<th>Event Listener</th>
<th>Receives</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrderStatusEventListener</td>
<td>OrderStatusVO events</td>
<td>Generated by trading interface adapters</td>
</tr>
<tr>
<td>PositionEventListener</td>
<td>PositionNutationVO events</td>
<td>Generated by the transaction service</td>
</tr>
<tr>
<td>QueryCacheEventListener</td>
<td>QueryCacheEvictionEventVO</td>
<td>Generated by the cache manager</td>
</tr>
<tr>
<td>QuoteEventListener</td>
<td>QuoteVO events (BidVO or AskVO)</td>
<td>Generated by market data interface adapters or fed from an external source</td>
</tr>
<tr>
<td>SessionEventListener</td>
<td>SessionEventVO events</td>
<td>Generated by market data and trading interface adapters</td>
</tr>
<tr>
<td>TickEventListener</td>
<td>TickVO events</td>
<td>Generated by market data interface adapters or fed from an external source</td>
</tr>
<tr>
<td>TradeEventListener</td>
<td>TradeVO events</td>
<td>Generated by market data interface adapters or fed from an external source</td>
</tr>
<tr>
<td>TransactionEventListener</td>
<td>TransactionVO events</td>
<td>Generated by the transaction service</td>
</tr>
<tr>
<td>AccountEventListener</td>
<td>AccountEventVO events</td>
<td>Generated by the account service</td>
</tr>
</tbody>
</table>

It is possible to change event listener priority, i.e. define the order in which the listeners are invoked when reacting to an event, by applying the following annotation to the listener method:

```java
@CrossOrigin
@EventHandlerPriority(value = EventHandlerType.EXECUTION, priority = 3)
public void onBar(BarVO bar) {
}
```

For `EventHandlerType` there are three options:

- **MARKET_DATA_CACHE**
- **DEFAULT**
- **EXECUTION**

MARKET_DATA_CACHE receives events first, and EXECUTION last.

Events for the same `EventHandlerType` are prioritized according to the `priority` value:

- **EVENT_HIGHEST_PRIORITY = 0**
- **EVENT_HIGH_PRIORITY = 1**
- **EVENT_NORMAL_PRIORITY = 2** (default value)
• EVENT_LOW_PRIORITY = 3
• EVENT_LOWEST_PRIORITY = 4
Again EVENT_HIGHEST_PRIORITY will receive events first, and EVENT_LOWEST_PRIORITY last.

Looking at a typical back test there are the following Bar consumers (in ascending order of event delivery):

• MarketDataCache (EventHandlerType = MARKET_DATA_CACHE, priority = EVENT_NORMAL_PRIORITY)
• StrategyService (your strategy) (EventHandlerType = DEFAULT, priority = EVENT_NORMAL_PRIORITY)
• Simulator (EventHandlerType = EXECUTION, priority = EVENT_NORMAL_PRIORITY)
So it is possible to change the order in which different listeners are invoked:

• To have StrategyService.onBar execute first add something like EventHandlerType = MARKET_DATA_CACHE, priority = EVENT_HIGH_PRIORITY)
• To execute StrategyService.onBar execute last add something like EventHandlerType = EXECUTION, priority = EVENT_LOW_PRIORITY)

11.6. JMS Destinations

Events are delivered to strategies via JMS. The following JMS Destinations are defined by the system

• one market data topic (MARKETDATA.TOPIC) defined inside applicationContext-server.xml: A Topic where all market data events are pushed into. On every event the securityId is set as a property, which can be used by the strategies to select market data events for securities subscribed to.

• one strategy queue per strategy (XXX.QUEUE) defined inside applicationContext-client-xxx.xml: A strategy specific Queue for messages like OrderStatus, Fills & Transactions. As JMS queues are persistent, messages will be delivered, even if a strategy was down, at the time of message creation.

• one generic topic (GENERIC.TOPIC) defined inside applicationContext-server.xml: A Topic for Generic Messages. Any strategy can send messages into this Topic. On every event the className of the event is set as a property, which can be used by the strategy to select event types that it is subscribed to.

Since market data events and generic events are pushed into two topics that are available to all strategies, strategies have to select appropriate messages on their own. This is the job of the SubscriptionService. It will modify the selectors on MessageListenerContainer accordingly and invoke the corresponding methods on the (server-side) MarketDataService (e.g. to request market data for additional securities).

**Important**

Strategies should never call the MarketDataService directly but instead call the SubscriptionService.
Configuration and Preferences API

12.1. Config Providers

AlgoTrader provides extensive support for configuration and customization of platform functions as well as of strategy specific settings.

The cornerstone of the configuration and preference API is the `ConfigProvider` interface that can be used to obtain arbitrary typed configuration parameters.

```java
class ConfigProvider {
    <T> T getParameter(String name, Class<T> clazz);
    Set<String> getNames();
}
```

`DefaultSystemConfigProvider` is the default implementation of `ConfigProvider` based on Spring `ConversionService` and is internally backed by a thread safe Map. The default provider makes use of `ConversionService` conversion framework to convert the content of the internal parameter map to the desired type. One can customize the process of parameter conversion by using a custom `ConversionService` implementation.

`ConfigParams` is a utility facade for `ConfigProvider` exposing a set of getter methods for common data types such as `Boolean`, `Integer`, `Long`, `Double`, `BigDecimal`, `URL` and `URI`. This class can be used by trading strategies that need to dynamically resolve configuration parameters at runtime.

`DefaultConfigLoader` is used to read configuration parameters from property files.

12.2. Config Beans

AlgoTrader also provides commonly used parameters in a form of plain Java beans referred to as Config beans. Common configuration are represented by `CommonConfig`. Core platform parameters are represented by `CoreConfig`. Instances of these classes are immutable and can be shared by multiple components and multiple threads of execution.

`ConfigBeanFactory` class can be used to create instances of Config beans based on configuration parameters using `@ConfigName` constructor parameter annotations. This factory is used to build standard `CommonConfig` and `CoreConfig` but it can also be used to build arbitrary Config beans for a trading strategy using the following convention.

```java
public final class StratConfig {
    private final String textParam;
}
```
private final boolean boolParam;
private final BigDecimal decimalParam;

public StratConfig(
    @ConfigName(value = "my.text") final String textParam,
    @ConfigName(value = "my.bool") final boolean boolParam,
    @ConfigName(value = "my.decimal", optional = true) final BigDecimal decimalParam) {
    this.textParam = textParam;
    this.boolParam = boolParam;
    this.decimalParam = decimalParam;
}

public String getTextParam() {
    return textParam;
}

public boolean isBoolParam() {
    return boolParam;
}

public BigDecimal getDecimalParam() {
    return decimalParam;
}

Each constructor parameter of a Config bean must be annotated with @ConfigName containing the parameter name. The config parameter type will be inferred from the constructor argument type. If a parameter is nullable and might be undefined in the config property files it can be marked as optional.

Standard platform Config beans such as CommonConfig and CoreConfig are declared in the Spring application context and get automatically injected into all beans that require configuration. One can also add strategy specific Config beans using the following bean definition:

```xml
<bean id="stratConfig" class="ch.algotrader.config.spring.ConfigBeanFactoryBean">
    <constructor-arg index="0" ref="configLocator" />
    <constructor-arg index="1" value="my.strategy.StratConfig" />
</bean>
```

Standard as well as strategy specific Config beans can be conveniently accessed using Spring SPEL expressions to wire other beans in the same Spring application context.

```xml
<bean id="MyObject" class="...">
    <constructor-arg value="${@stratConfig.requestUri}" />
</bean>
```
In addition it is possible to reference individual beans (e.g. config beans) directly within Spring wired classes.

```java
private @Value("#{@configParams.accountId}") long accountId;
```

### 12.3. Config Locator

Even though it is preferable to dependency injection services provided by Spring application context to obtain configuration details required by custom components, in certain cases it may be necessary for unmanaged beans to get hold of Config beans. This can be done through the global `ConfigLocator`.

```java
ConfigParams configParams = ConfigLocator.instance().getConfigParams();
CommonConfig commonConfig = ConfigLocator.instance().getCommonConfig();
StratConfig stratConfig = ConfigLocator.instance().getConfig(StratConfig.class);
```
Chapter 13. Processes and Networking

13.1. Processes

The following Services and Process are used by the system:

Table 13.1. Services and Processes

<table>
<thead>
<tr>
<th>Service / Process</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlgoTrader Server</td>
<td>This is the main AlgoTrader process</td>
</tr>
<tr>
<td>Strategies</td>
<td>In Live Trading Mode each strategy can run in its own Java process or within the same process as the AlgoTrader Server (one strategy only). In simulation mode, the strategies run within the same process as the AlgoTrader Server</td>
</tr>
<tr>
<td>MySql</td>
<td>Main database process. Alternatively to using MySql the embedded in-memory database H2 can be run within the process of the AlgoTrader Server in simulation mode.</td>
</tr>
<tr>
<td>InfluxDB</td>
<td>The InfluxDB database used for storage of historical data</td>
</tr>
</tbody>
</table>

If the AlgoTrader Server and the strategies are running within separate processes, individual strategies can be stopped / altered / restarted independent of each other and the AlgoTrader Server.

13.2. Sockets

Table 13.2. Sockets

<table>
<thead>
<tr>
<th>Socket</th>
<th>Description</th>
<th>Default Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI Exporter</td>
<td>Remote access to Spring services</td>
<td>1199</td>
</tr>
<tr>
<td>ActiveMQ TCP</td>
<td>ActiveMQ TCP transport</td>
<td>61616</td>
</tr>
<tr>
<td>ActiveMQ WS</td>
<td>ActiveMQ WebSockets transport</td>
<td>61614</td>
</tr>
<tr>
<td>ActiveMQ WSS</td>
<td>ActiveMQ Secure WebSockets transport</td>
<td>61613</td>
</tr>
<tr>
<td>ActiveMQ Stomp</td>
<td>ActiveMQ TCP Stomp transport</td>
<td>61617</td>
</tr>
<tr>
<td>ActiveMQ SSL Stomp</td>
<td>ActiveMQ Secure TCP Stomp transport</td>
<td>61618</td>
</tr>
<tr>
<td>HTTP</td>
<td>Jetty HTTP transport</td>
<td>9090</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Jetty HTTPS transport</td>
<td>9443</td>
</tr>
<tr>
<td>MySql</td>
<td>MySql database connection</td>
<td>3306</td>
</tr>
<tr>
<td>InfluxDB</td>
<td>InfluxDB connection</td>
<td>8086</td>
</tr>
<tr>
<td>IB Gateway</td>
<td>defined by IB Gateway configuration</td>
<td>4001</td>
</tr>
<tr>
<td>Bloomberg Terminal</td>
<td>BBComm.exe</td>
<td>8194</td>
</tr>
<tr>
<td>Fix</td>
<td>Fix Connections</td>
<td>varies</td>
</tr>
</tbody>
</table>
13.3. RMI

The system defines an RMI services through Spring Remoting (RMI Registry 1199): RmiServiceExporter (defined in applicationContext-export-remoteServices.xml)
Chapter 14.

Hibernate Sessions and Caching

AlgoTrader uses Hibernate for accessing and persisting objects to the database.

14.1. Hibernate Caching

In order to prevent having to access the database on every single request, Hibernate provides two types of caches:

First Level Cache
The First Level Cache is always associated with the current Session. Hibernate uses this cache by default. Its main purpose is to reduce the number of SQL queries needed to execute within a given transaction. Instead of updating after every modification done to an object separately, it updates the database only once at the end of the transaction. At the end of a Session, the attached First Level Cache will be destroyed. In case an Object, that is already loaded into the current First Level Cache, is modified outside the Session, the First Level Cache will not get notified of the change.

Second Level Cache
The Second Level Cache is always associated with the Session Factory. While processing transactions, Hibernate stores objects at the Session Factory level, so that those objects will available to the entire system. Whenever a new query is executed, Hibernate will first check with the Second Level Cache to see whether the objects are available in the Cache.

Both First and Second Level Cache require a Hibernate Session. Creation of a Session is usually very quick (a few milliseconds). This mechanism is therefore fine for any request-response based system. However this approach is not feasible for a trading application. A trading application typically receives several thousand market data events per second. Ideally these market data events have to be matched to the latest data stored in the database (e.g. Security related information, current Positions, executed Trades, etc.). Opening a new Hibernate Session for every market data event, to synchronize related objects (like corresponding Security), is much too expensive!

For this purpose AlgoTrader introduces a Level-Zero Cache

14.1.1. Level-Zero Cache

AlgoTrader Level-Zero Cache is an additional Caching Level on top of Hibernate First and Second Level Cache which has the following features:

- Level-Zero Cache is a pure Java based Cache
- Level-Zero Cache does not require an active Hibernate Session
- Objects available inside the Level-Zero Cache will be delivered instantaneously and do not introduce any additional latency
- Level-Zero Cache does refresh objects at the same time a database update occurs
• Level-Zero Cache is a read-only Cache, it does not provide any sort update functionality. Changes to Entities retrieved from the Level-Zero Cache will never be persisted to the database. In order to modify objects in the database Spring services and Hibernate DAOs have to be used.

• Level-Zero Cache preserves object identity, so graphs and cyclical references are allowed. Therefore objects retrieved from the Level-Zero Cache can be compared using the `equals()` method but also using the comparison operator `==`.

• Level-Zero Cache provides ad-hoc initialization of Hibernate Proxies and Persistent Collections. Newly initialized Proxies and Persistent Collections will be added to the Cache automatically.

• Level-Zero Cache does not provide any Passivation or Eviction. All Objects stay in memory. It is therefore not recommended to use the Level-Zero Cache for Objects that are only needed one time, especially if there is a large number of those objects.

• Level-Zero Cache is available to both the JVM containing the AlgoTrader Server as well as all Strategy JVMs.

• No Proxies, no Byte Code Instrumentation and no Annotations are needed for Level-Zero Cache to work.

By using the Level-Zero Cache it is possible to work on fully up-to-date versions of Entities without introducing any latency penalties.

Access to the Level-Zero Cache is provided by the class `ch.algotrader.cache.CacheManagerImpl` which is exposed as a Spring Bean named `cacheManager`. The `CacheManagerImpl` provides these public methods to access the Level-Zero Cache:

- `get` gets an Entity of the given `clazz` by the defined `id`.
- `getAll` gets all Entities of the given `clazz`.
- `contains` checks whether an Entity of the given `clazz` and `id` is in the Cache.
- `initializeProxy` lazy-initializes the given `key` of the specified `entity`.
- `initializeCollection` lazy-initializes the given `collection` of the specified `entity`.
- `find` performs the given HQL query by passing defined `namedParameters`.
- `find` performs the given HQL query by passing defined `maxResults` (passing zero will return all elements) `namedParameters`.
- `findUnique` performs the given unique HQL query by passing defined `namedParameters`.
clear

clears the entire cache

Like Hibernate First and Second Level Cache the AlgoTrader Level-Zero Cache will first check if the requested object is available inside the Cache. If not, the object will be retrieved via the ch.algotrader.hibernate.GenericDao and stored in the Cache.

The class EntityCache is responsible for caching of Entities (handled by the EntityHandler) and Entity-Collections (handled by CollectionHandler). When adding a new Entity to the cache, the EntityHandler and CollectionHandler traverse the entire object graph of initialized Entities and Collections and store all of them in separate nodes of the internal Hash Map Cache. Java reflection is used for the object graph traversal (on new objects or object updates).

The class QueryCache is responsible for caching of Hibernate Query Results. Query Results are cached in reference to all involved tables. So whenever one of those referenced tables is modified, the corresponding Query Cache Entry is removed (detached).

Through the class ch.algotrader.wiring.server.CacheWiring net.sf.ehcache.event.CacheEventListenerAdapter are registered with each Ehcache element. These are either EntityCacheEventListener for Entity caches or CollectionCacheEventListener for Collection Caches. These EventListeners emit CacheEvictionEvents via the AlgoTrader EventDispatcher to CacheManager of the Server and Strategies (running in remote JVMs) to get notified every time an Entity / Collection is updated or the UpdateTimestampsCache (related to the Hibernate StandardQueryCache) has been updated.

Note

The LookupService uses the CacheManager for all lookup operations instead of using Hibernate DAOs directly.
Logging

15.1. Custom UI Log Event Appender

This is a special customized appender which allows to send log events to the UI. The log events are sent via JMS/STOMP and log levels and loggers are configurable. For example you could define a particular logger (e.g. some specific class) or use the Root logger configured for the desired log level, e.g. INFO or WARN. In order to have multiple loggers with multiple log levels, separate appenders must be created each with it's own filter. The sample configuration below will create two UI appenders - one with level WARN (and above), another with INFO. Loggers defined in "Loggers" section reference these appenders in such a way that INFO log entries from PortfolioServiceImpl as well as all the entries with level WARN and above will be sent to the UI.

Log entries are wrapped inside JMS message and get propagated via WebSocket STOMP protocol to the UI. In order to consume the log message the client will have to be subscribed to specific JMS topic ("/topic/log-event"). See StompAPIUtils.js in the HTML client for the subscription logic example.

```xml
<Appenders>
  <LogEvent name="LogEventWARN">
    <ThresholdFilter level="WARN" onMatch="ACCEPT" onMismatch="DENY"/>
  </LogEvent>
  <LogEvent name="LogEventINFO">
    <ThresholdFilter level="INFO" onMatch="ACCEPT" onMismatch="DENY"/>
  </LogEvent>
</Appenders>

<Loggers>
  <Root level="debug">
    <AppenderRef ref="LogEventWARN"/>
  </Root>
  <Logger name="ch.algotrader.service.PortfolioServiceImpl">
    <AppenderRef ref="LogEventINFO" />
  </Logger>
</Loggers>
```